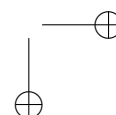
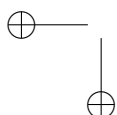
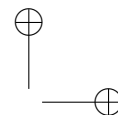
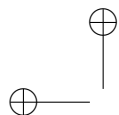


DESTRIPANDO INTERNET

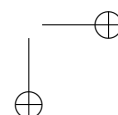
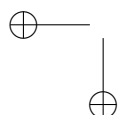
6 de mayo de 2026

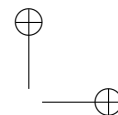
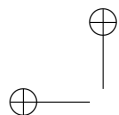




DESTRIPANDO INTERNET

David Villa
Ana Rubio





© 2026 David Villa, Ana Rubio

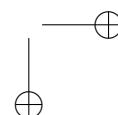
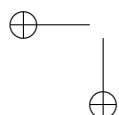
Primera edición: 2026

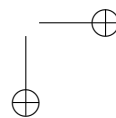
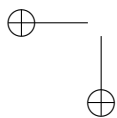
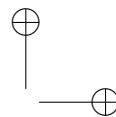
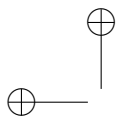
ISBN: XXX-XX-XXXXX-X

e-mail `destripando.internet@gmail.com`

web `https://github.com/destripando-internet`

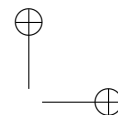
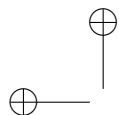
© Los autores del documento. Se permite la copia, distribución y/o modificación de este documento bajo los términos de la licencia de documentación libre GNU, versión 1.1 o cualquier versión posterior publicada por la *Free Software Foundation*, sin secciones invariantes. Puede consultar esta licencia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





Índice general

Prefacio	xxiii
1 Debian GNU/Linux	1
1.1. UNIX	1
1.2. El fin del software libre	2
1.3. GNU	3
1.4. Linux	4
1.5. GNU/Linux	4
1.6. Debian GNU/Linux	5
2 Shell	9
2.1. GNU Bash	10
2.2. Valor de retorno	11
2.3. Opciones	12
2.4. Procesos	13
2.4.1. Trabajos	14
2.4.2. Otros modos de identificar procesos.	15
2.5. Entrada, salida y salida de error	16
2.6. Redirección	17
2.7. Usuarios y grupos	21
2.7.1. Propietarios y permisos	22
2.8. Paquetes	24
2.8.1. Repositorios de paquetes	25
2.8.2. Cómo encontrar paquetes	27



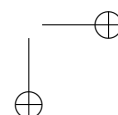
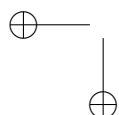
2.9. Servicios	28
2.10. Parámetros del núcleo	30

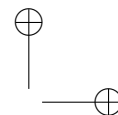
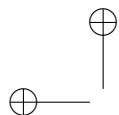
3 Python 33

3.1. ¡A programar!	33
3.2. Variables y tipos	34
3.3. Tipos de datos	34
3.3.1. Valor nulo	34
3.3.2. Booleanos	35
3.3.3. Numéricos	35
3.3.4. Secuencias	35
3.4. Módulos.	37
3.5. Estructuras de control.	37
3.6. Indentación estricta	38
3.7. Funciones	38
3.8. Python <i>is different</i>	39
3.9. Hacer un ‘ejecutable’	39
3.10. Orientado a objetos	40
3.11. <i>Type checking</i>	40

4 Internet 43

4.1. Tecnología Internet	44
4.2. Protocolos	45
4.3. Pila de protocolos	47
4.4. Modelo OSI	48
4.4.1. Direccionamiento físico vs. lógico	49
4.5. Modelo TCP/IP	49
4.6. Modelo híbrido.	50
4.7. Qué no es Internet	51
4.7.1. Internet no es la web	51





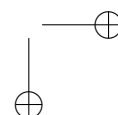
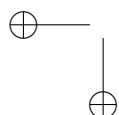
4.7.2.	Internet no es la nube	51
4.7.3.	Internet no es TCP/IP	51
4.7.4.	Internet no es un empresa u organización	52

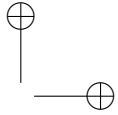
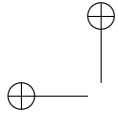
5	Protocolos esenciales	53

5.1.	Encapsulación	54
5.2.	Protocolos de enlace	57
5.3.	Ethernet	59
5.3.1.	Interfaces de red	59
5.3.2.	Trama Ethernet	60
5.3.3.	Direcciones MAC	61
5.3.4.	Conectividad Ethernet	63
5.4.	PPP	64
5.5.	IP	65
5.5.1.	Interfaces de red	65
5.5.2.	Paquete IP	66
5.5.3.	Direcciones IP	68
5.5.4.	Conectividad IP	69
5.6.	ARP	69
5.7.	ICMP	72
5.7.1.	Conectividad IP	74
5.8.	UDP	76
5.8.1.	Datagrama UDP	77
5.8.2.	Conectividad UDP	79
5.9.	TCP	79
5.9.1.	Segmento TCP	80
5.9.2.	Capturando una conexión TCP	82
5.9.3.	Conectividad TCP	83

6	Sockets	85

6.1.	Programación de redes	86

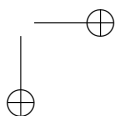
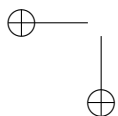




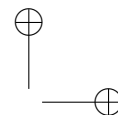
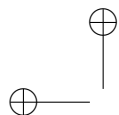
6.2. La clase <code>socket</code>	86
6.3. Puertos	87
6.4. Comunicación UDP	89
6.5. Servidor TCP	92
6.6. Cliente TCP	94
6.7. Flujos de datos y E/S parcial	95
6.7.1. Envío TCP	96
6.7.2. Recepción TCP	97
6.7.3. Bandera de fin de mensaje	98
6.7.4. Tamaño del mensaje en la cabecera	100
6.8. Sockets como archivos	101
6.9. Gestión explícita de buffers	103
6.10. Fin de la comunicación TCP	104
6.11. Manejo de errores	105
6.11.1. Context manager	107
6.12. Netcat	108
6.12.1. Sintaxis básica	109

7 Direccionamiento IP 115

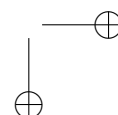
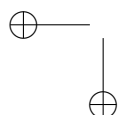
7.1. Qué es una dirección IP y su formato	115
7.2. Máscara de red	116
7.3. Direccionamiento con clases	117
7.3.1. <i>Subnetting</i>	118
7.3.2. Ejemplo de <i>subnetting</i>	119
7.4. Direccionamiento sin clases	120
7.4.1. VLSM	121
7.4.2. Bloques /30	121
7.4.3. Ejemplo de VLSM	122
7.5. Direcciones especiales	126

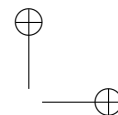
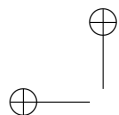


8 Interconexión	129
8.1. Almacenamiento y reenvío	130
8.2. Entrega directa vs. indirecta	133
8.3. Tabla de encaminamiento	134
8.3.1. Analizando la tabla de ejemplo	136
8.3.2. Tabla de nodo final	137
8.4. Agregación	138
8.5. Un par de ejemplos detallados	140
8.5.1. Escenario 1: entrega local	140
8.5.2. Escenario 2: dos saltos	142
8.6. Mensajes de control de interred (ICMP)	143
8.6.1. Destino inalcanzable (<i>Destination unreachable</i>) . . .	145
8.6.2. Tiempo excedido (<i>Time exceeded</i>).	147
8.6.3. Problema en parámetro (<i>Parameter problem</i>)	148
8.6.4. Supresión al origen (<i>Source quench</i>)	148
8.6.5. Redirección (<i>Redirect</i>)	148
8.6.6. Ping (<i>Echo</i>)	149
8.6.7. Marca de tiempo (<i>Timestamp</i>)	151
8.6.8. Información (<i>Information</i>) y máscara (<i>Address mask</i>)	152
8.6.9. Solicitud y anuncio de router	152
8.7. Fragmentación	152
8.7.1. No fragmentar.	161
8.7.2. Descubrimiento de MTU	162
9 Encaminamiento dinámico	165
9.1. Algoritmos y protocolos de encaminamiento	166
9.2. Sistemas autónomos	167
9.3. Topología de referencia: triple delta	168
9.4. Redundancia	169
9.5. La ruta más corta	170



9.6. Vector-distancia	171
9.6.1. Problemas y limitaciones de vector-distancia	174
9.6.2. RIP	175
9.7. Estado de enlace	176
9.7.1. OSPF	177
9.8. Vector-ruta y BGP	177
9.9. Laboratorio de encaminamiento Delta	179
9.9.1. Encaminamiento estático.	183
9.9.2. Zebra.	184
9.9.3. Encaminamiento RIP	185
9.9.4. Encaminamiento OSPF	188
9.9.5. Reacción ante fallos.	189
10 Configuración IP	193
10.1. Configuración manual	193
10.2. Configuración automática con DHCP	195
10.3. Servidor DHCP	199
11 Confiabilidad y control de flujo	203
11.1. Parada y espera	205
11.2. Repetición continua	206
11.3. Repetición selectiva	208
11.4. Confiabilidad en comunicaciones duplex.	210
12 Confiabilidad y control de flujo en TCP	211
12.1. Conexión	212
12.2. Tamaño máximo de segmento	214
12.3. Desconexión	216
12.3.1. Tiempo de silencio	218
12.4. Reset	218
12.5. Control de flujo en TCP	220

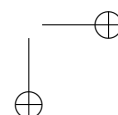
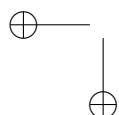


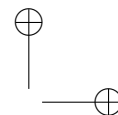
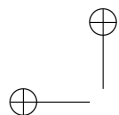


12.6. Ventanas de envío y recepción	222
12.7. RTO: el temporizador de retransmisión	224
12.8. El síndrome de la <i>ventana tonta</i>	229
12.9. Cierre de la ventana	230
12.10. Confirmación selectiva	230
12.11. Aplicaciones interactivas	232
12.12. Control de errores	233
12.12.1. Segmento perdido/corrupto	233
12.12.2. Confirmación perdida/corrupta	233
12.12.3. Segmento duplicado	234
12.12.4. Segmento fuera de orden	235
12.12.5. Confirmación duplicada	236
12.12.6. Demasiadas retransmisiones	237
12.13. <i>Keep alive</i>	237

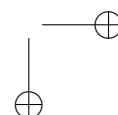
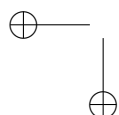
13 Control de congestión 239

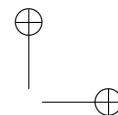
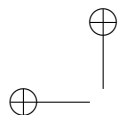
13.1. Carga, capacidad y congestión	239
13.2. Control de congestión	242
13.2.1. Técnicas preventivas	242
13.2.2. Técnicas reactivas.	243
13.3. Control de flujo vs. control de congestión	244
13.4. Supresión al origen.	244
13.5. Control de congestión en TCP	245
13.5.1. Arranque lento (<i>slow start</i>)	246
13.5.2. Evitación de congestión (<i>congestion avoidance</i>)	247
13.6. Decrecimiento multiplicativo	248
13.6.1. Indicios de congestión	249
13.6.2. Modelo simplificado.	251
13.7. Gestión activa de colas	253
13.7.1. ECN en TCP	254
13.7.2. ECN con otros protocolos	256



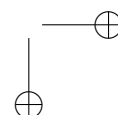
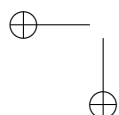


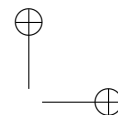
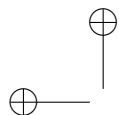
14 Cliente-Servidor	259
14.1. Upper	261
14.2. Servidor TCP	261
14.3. Cliente TCP	263
14.4. Servidor TCP multihilo	265
14.5. Forzando los límites	266
14.6. Servidor TCP multiproceso	267
14.7. Servidor UDP	271
14.8. Servidor UDP multiproceso	274
14.9. Servidor UDP multihilo	276
14.10.socketserver.	277
14.11.Operaciones bloqueantes	279
14.12.Alternativas a la E/S bloqueante	279
14.13.Servidor TCP asíncrono con <code>select()</code>	281
14.14.Servidor TCP asíncrono con <code>asyncio</code>	283
14.15.Cliente TCP asíncrono con <code>asyncio</code>	286
14.16.Servidor UDP asíncrono con <code>asyncio</code>	287
 15 Publicador-Suscriptor	 291
15.1. Chat UDP para dos	292
15.1.1. Paso 1: Mensaje unidireccional	292
15.1.2. Paso 2: Devuelve el saludo	294
15.1.3. Paso 3: Libertad de expresión	294
15.1.4. Paso 4: Habla cuando quieras	296
15.1.5. Paso 5: Todo en uno	298
15.2. Chat asíncrono con <code>select()</code>	299
15.3. Chatroom UDP	301
15.4. Chatroom TCP	303
15.5. Chatroom con MQTT	305



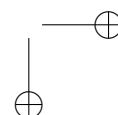
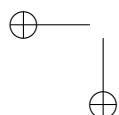


16 Calidad de servicio	309
16.1. Tasa de transferencia	310
16.1.1. Tasa instantánea	311
16.1.2. Promedio acumulado (CA)	311
16.1.3. Media móvil simple (SMA)	312
16.1.4. Media móvil exponencial (EMA)	313
16.1.5. Consideraciones sobre el cálculo de tasa	314
16.2. Control de flujo TCP como limitador de tasa	315
16.3. Limitación de tasa en el receptor	316
16.4. Limitación de tasa en el emisor	321
 17 Serialización	 323
17.1. Representación, sólo eso	324
17.2. Los enteros de Python	325
17.3. Caracteres	326
17.4. Tipos multibyte y ordenamiento	328
17.5. Cadenas de caracteres y secuencias de bytes	330
17.6. Empaquetado	332
17.7. Desempaquetado	334
17.8. Formatos de serialización binaria	334
17.9. Serialización textual	335
 18 Captura y análisis	 337
18.1. tshark	338
18.1.1. Acceso privilegiado	339
18.1.2. Selección de la interfaz de captura	339
18.1.3. Limitando la captura	341
18.1.4. Filtros de captura	341
18.1.5. Formato de salida	341
18.1.6. Filtros de visualización	344
18.1.7. Estadísticas	345





18.2. wireshark	348
18.2.1. Interfaz gráfica	348
18.2.2. Captura y filtrado	350
18.3. Captura de flujos.	353
19 Sockets raw	357
19.1. Acceso privilegiado	358
19.2. Tipos de sockets raw.	359
19.3. Sockets AF_PACKET:SOCK_RAW	359
19.3.1. Construir y enviar tramas	361
19.3.2. Implementando un arping	362
19.4. Sockets AF_INET:SOCK_RAW.	365
19.4.1. Capturando mensajes.	365
19.4.2. Enviando.	366
20 Redes Privadas	369
20.1. Líneas alquiladas.	369
20.2. Redes privadas TCP/IP	371
20.2.1. Direccionamiento privado	371
20.3. Conectividad en redes privadas.	372
20.3.1. Traducción de Direcciones de Red (NAT).	374
20.4. Reenvío de puertos.	377
21 DNS	379
21.1. Dominios y zonas	382
21.2. La zona raíz	384
21.3. Resolución de nombres	385
21.4. Cachés e información obsoleta	388
21.5. Configuración de zona.	390
21.6. Resolución inversa	391
21.7. Replicación	391
21.7.1. CDN	392



21.8. Transporte	393
21.9. Multicast DNS	393
21.10. Dynamic DNS	394
21.11. Servidores DNS <i>sinkhole</i>	395

22 SSH 397

22.1. Shell segura.	397
22.2. Configuración.	398
22.3. Acceso con clave pública	398
22.4. Autenticación con certificado	400
22.4.1. Un certificado para el usuario	400
22.5. Copia de archivos con SCP	402

23 Epílogo 403

A Comandos habituales 407

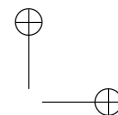
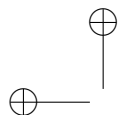
A.1. Ficheros y directorios	407
A.2. Sistema	409
A.3. Procesos	410
A.4. Usuarios y permisos	410

B Otros comandos de red 411

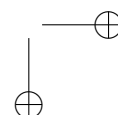
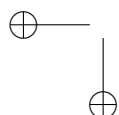
B.1. Otros <i>ping</i>	411
B.2. <i>mtr</i>	411
B.3. <i>ss</i>	412

C Docker 415

C.1. Imágenes	415
C.2. Contenedores	416
C.3. <i>dockerfile</i>	418
C.4. Docker Compose	418



D Unidades	421
D.1. Memoria	421
D.2. Almacenamiento	421
D.3. Tasa o velocidad de transmisión	422
Referencias	429



Listado de acrónimos

8P8C	Eight Position Eight Contact
A	Address
AAAA	Quad-A; DNS record for IPv6 addresses
ABR	Area Border Router
ACK	Acknowledgement
ADSL	Asymmetric Digital Subscriber Line
ANSI	American National Standards Institute
API	Application Program Interface
AQM	Active Queue Management
ARP	Address Resolution Protocol
ARQ	Automatic Repeat reQuest
AS	Autonomous System
ASPATH	AS-PATH
ASBR	Autonomous System Border Router
ASCII	American Standard Code for Information Interchange
ASIC	Application-Specific Integrated Circuit
ASN	Abstract Syntax Notation
ATM	Asynchronous Transfer Mode
BGP	Border Gateway Protocol
BOOTP	Bootstrap Protocol
BSD	Berkeley Software Distribution
CA	Certificate Authority
CA	Cumulative Average
CAD	Computer-Aided Design
CAN	Control Area Network
CD	Continuous Delivery
CDN	Content Delivery Network
CE	Congestion Experienced
CHAP	Challenge Handshake Authentication Protocol
CI	Continuous Integration

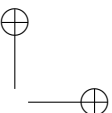
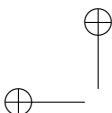
CIDR	Classless Interdomain Routing
CLI	Command Line Interface
CNAME	Canonical Name
CPU	Central Processing Unit
CR	Carriage Return
CRC	Cyclic Redundancy Check
CSS	Cascading Style Sheets
CSV	Comma Separated Values
CWR	Congestion Window Reduced
D-BUS	Desktop BUS
DDNS	Dynamic DNS
DF	Don't Fragment
DHCP	Dynamic Host Configuration Protocol
DNAT	Destination NAT
DNS	Domain Name System
DNS-SD	DNS Service Discovery
DoH	DNS over HTTP
DoQ	DNS over QUIC
DoS	Denial of Service
DoT	DNS over TLS
eBGP	External BGP
ECE	ECN Echo
ECMP	Equal-Cost Multi-Path
ECN	Explicit Congestion Notification
ECT	ECN Capable Transport
EGP	Exterior Gateway Protocol
EMA	Exponential Moving Average
EOL	End Of Line
ESI	Escuela Superior de Informática
E/S	Entrada/Salida
FCS	Frame Check Sequence
FQDN	Fully Qualified Domain Name
FLSM	Fixed Length Subnet Mask
FPGA	Field-Programmable Gate Array
FRR	Free Range Routing
FTP	File Transfer Protocol
FTTH	Fiber To The Home
FSF	Free Software Foundation

GID	Group IDentifier
GIL	Global Interpreter Lock
GNOME	GNU Network Object Model Environment
GNU	GNU is Not Unix
GPL	General Public License
GSM	Global System for Mobile Communications
GUI	Graphical User Interface
HDLC	High-Level Data Link Control
HFC	Hybrid Fiber-Coaxial
HTTP	Hypertext Transfer Protocol
HTTP/2	HTTP versión 2
HTTP/3	HTTP versión 3
HTTPS	HTTP sobre SSL
Hurd	Hird of Unix-Replacing Daemons
IANA	Internet Assigned Numbers Authority
iBGP	Internal BGP
IBM	International Business Machines
ICANN	Internet Corporation for Assigned Names and Numbers
ICMP	Internet Control Message Protocol
IEEE	Institute of Electrical and Electronics Engineers
IETF	Internet Engineering Task Force
IGMP	Internet Group Management Protocol
IGP	Interior Gateway Protocol
IHL	Internet Header Length
IMAP	Internet Message Access Protocol
I/O	Input/Output
IoT	Internet of Things
IP	Internet Protocol
IPC	Inter-Process Communication
IPTV	IP Television
IPv4	IP versión 4
IPv6	IP versión 6
IPX	Internetwork Packet Exchange
IRDP	ICMP Router Discovery Protocol
ISN	Initial Sequence Number
ISO	International Organization for Standardization
ISP	Internet Service Provider
IS-IS	Intermediate System to Intermediate System

JSON	JavaScript Object Notation
KiB	Kibibyte (1 024 bytes)
LAN	Local Area Network
LCP	Link Control Protocol
LED	Light Emitting Diode
LF	Line Feed
LPM	Longest Prefix Match
LSA	Link-State Advertisement
LSB	Least Significant Bit
LSDB	Link State Database
LTE	Long Term Evolution
MD5	Message-Digest Algorithm 5
mDNS	Multicast DNS
MiB	Mebibyte (1 024 KiB)
MAC	Media Access Control
MIT	Massachusetts Institute of Technology
MF	More Fragments
MitM	Man in the Middle
MQTT	Message Queue Telemetry Transport
MSB	Most Significant Bit
MSL	Maximum Segment Lifetime
MSS	Maximum Segment Size
MTU	Maximum Transmission Unit
MX	Mail eXchange
NACK	Negative Acknowledgement
NAPT	Network Address Port Translation
NAT	Network Address Translation
NCP	Network Control Protocol
NIC	Network Interface Controller
NS	Name Server
ONT	Optical Network Terminal
OSI	Open Systems Interconnection
OSPF	Open Short Path First
OUI	Organizationally Unique Identifier
P2P	Peer to Peer
PAP	Password Authentication Protocol
PC	Personal Computer
PID	Process IDentifier

PLPMTUD	Packetization Layer Path MTU Discovery
PMTU	Path MTU
PMTUD	Path MTU Discovery
POO	Programación Orientada a Objetos
PoP	Point of Presence
POP	Post Office Protocol
POSIX	Portable Operating System Interface; UNIX
PPP	Point to Point Protocol
PTR	Pointer
QoS	Quality of Service
QUIC	Quick UDP Internet Connections
RAE	Real Academia Española
RD	Recursion Desired
REST	REpresentational State Transfer
RED	Random Early Detection
RFC	Request For Comments
RIP	Routing Information Protocol
RIR	Regional Internet Registry
RJ45	Registered Jack 45
ROM	Read Only Memory
RSA	Rivest, Shamir y Adleman
RTO	Retransmission TimeOut
RTP	Real-time Transport Protocol
RTT	Round-Trip Time
RTTVAR	Round-Trip Time Variation
SA	Sistema Autónomo
SACK	Selective Acknowledgement
SCP	Secure Copy Protocol
SCTP	Stream Control Transmission Protocol
SDSL	Symmetric Digital Subscriber Line
SI	Sistema de Información
SIP	Session Initiation Protocol
SMA	Simple Moving Average
SMTP	Simple Mail Transport Protocol
SNAT	Source NAT
SND.UNA	Send Unacknowledged Number
SO	Sistema Operativo
SOA	Service Oriented Architecture

SPT	Shortest Path Tree
SRT	Service Response Time
SRTT	Smoothed Round-Trip Time
SRV	Service
SSH	Secure SHell
SSL	Secure Socket Layer
SYN	Synchronization
SWS	Silly Window Syndrome
TFTP	Trivial File Transfer Protocol
TCB	Transmission Control Block
TCP	Transmission Control Protocol
TCP/IP	Arquitectura de protocolos de Internet
TIC	Tecnologías de la Información y las Comunicaciones
TLD	Top Level Domain
TLS	Transport Layer Security
TOML	Tom's Obvious, Minimal Language
ToS	Type of Service
TTL	Time To Live
TTY	TeleTYpewriter
TXT	Text
UCLM	Universidad de Castilla-La Mancha
UID	User Identifier
UDP	User Datagram Protocol
URI	Universal Resource Identifier
URL	Uniform Resource Locator
USB	Universal Serial Bus
UTF	Unicode Transformation Format
UTF8	UTF de 8 bits
UUID	Universally Unique Identifier
VLSM	Variable Length Subnet Mask
VM	Virtual Machine
VoIP	Voice over IP
VPN	Virtual Private Network
WAN	Wide Area Network
WiFi	Wireless Fidelity
WWW	World Wide Web
XML	Extensible Markup Language
YAML	YAML Ain't Markup Language
ZWP	Zero Window Probe



Prefacio

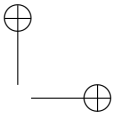
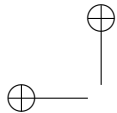
Las redes de comunicaciones, y especialmente Internet, se convirtieron ya hace décadas en una parte esencial de nuestras vidas. Internet cambió radicalmente nuestra forma de comprar, viajar, hacer negocios, y relacionarnos. Fue un cambio mucho más importante que la invención de la imprenta o la revolución industrial, supuso un nuevo concepto de comunicación, un medio de difusión de información y conocimiento a una escala sin precedentes. Y es precisamente la comunicación lo que nos convierte en seres sociales, en humanos. Las redes cambiaron cómo vivimos y lo que somos.

Todo ello es motivo suficiente para que cualquier persona (y en particular cualquier técnico) se interese por el funcionamiento de las redes de computadores. Entender cómo son y cómo funcionan las redes ayuda a tomar conciencia de sus posibilidades y oportunidades, y también de sus riesgos.

Destripando Internet

Según la RAE una de las acepciones de *destripar* es ‘sacar lo interior de algo’; y eso vamos a hacer aquí, sacar y estudiar las tripas de Internet: la tecnología TCP/IP que la hace posible, los mecanismos y los protocolos clave en su funcionamiento. La mejor manera de conseguir esto es con un enfoque muy práctico, y por suerte, mucha de la tecnología de Internet es sorprendentemente accesible y está disponible para cualquiera. Esto es así en buena medida por la naturaleza abierta de las normas y protocolos que la rigen y por el gran protagonismo del software libre en este ámbito. Basaremos este enfoque práctico 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. Internet y toda la tecnología que la hace posible nació en Unix. GNU es heredero de todo ese legado.
- GNU está disponible para cualquiera, en cualquier parte, sin ninguna limitación. Es la esencia del *software libre*, 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 extensa comunidad de usuarios de GNU y todas sus distribuciones derivadas también es uno de sus puntos fuertes. Los sistemas operativos GNU/Linux y el software libre en general son las opciones más utilizadas con mucho en servidores y en equipos de comunicaciones en todo el mundo.

El entorno de trabajo y herramientas que se utilizan en este libro están disponibles en el sistema operativo Debian GNU/Linux [1]. Por simplicidad y para un máximo aprovechamiento, te aconsejamos instalar esta distribución en tu computador, ya que este será el entorno que asumiremos cuando se hable de configuración o se mencionen comandos del sistema. Todos los programas que emplearemos están disponibles como paquete oficial y se pueden instalar con el gestor de paquetes `apt` sin necesidad de configuración específica.

Por supuesto, cualquier distribución basada en GNU podría ser perfectamente válida para la realización de los ejemplos y ejercicios prácticos propuestos, aunque quizá podrías encontrar ciertas diferencias. Simplemente ten en cuenta, que en ausencia de indicaciones concretas, debes asumir que hablamos de Debian o sus derivados, siendo Ubuntu el más popular.

También asumimos que dispones de un computador con al menos una interfaz de red Ethernet o WiFi con acceso a Internet convenientemente configurada y también que tienes una cuenta de usuario con privilegios de administrador en tu propio computador.

Python

Python es un lenguaje de programación moderno y muy popular. 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++, o C#. Python ha conseguido una gran notoriedad en los últimos años por su potencia y versatilidad en el manejo de datos (análisis, visualización, etc.) y por su amplio uso en muchas de las tecnologías relacionadas con la inteligencia artificial. Pero Python va mucho más allá.

Una de las características que lo hace especialmente adecuado para la «programación de redes» es que su librería estándar respeta la nomenclatura y conceptos del API de programación de POSIX, lo que resulta muy útil para encontrar documentación y bibliografía, además de facilitar la *traducción* a Python de las llamadas al sistema de POSIX, que están implementadas en C. Cuando en el texto hagamos referencia a una llamada al sistema usaremos el superíndice ^{sc} ‘*system call*’, como por ejemplo en: `write()`^{sc}.

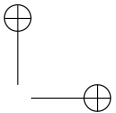
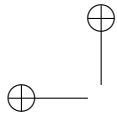
Python también es software libre, lo que significa que está disponible para cualquiera en cualquier plataforma. Está respaldado por una enorme comunidad de usuarios y desarrolladores, y suele ser muy fácil encontrar ayuda, incluso en temas muy específicos.

En todos los ejemplos, listados de código y programas que se proporcionan junto con el libro, utilizaremos siempre Python 3.

Contenido del libro

La mayoría de los libros de redes y comunicaciones suelen tener una estructura similar. Eso se debe a que las redes se estudian, e incluso implementan, mediante los denominados modelos de referencia, especialmente el «modelo OSI». Este modelo divide todo el conocimiento relacionado con las comunicaciones en redes en una serie de capas dispuestas como una pila. La primera (la inferior) es la capa física y se ocupa de los detalles de más bajo nivel llegando a los valores de voltaje o el tipo de cableado. La última es la capa de aplicación y se ocupa del más alto nivel, como por ejemplo, la estructura de un *e-mail*. Los autores suelen seguir este modelo ya sea empezando por la inferior hacia arriba o desde la superior hacia abajo, explicando las tecnologías y funcionamiento de cada componente capa a capa.

El problema con este enfoque es que esas capas no son ni de lejos tan independientes como se dice. Hay muchos mecanismos esenciales que requieren entender los detalles de lo que ocurre en otras capas. A nosotros no nos



parece buena idea ignorarlos o postergar su explicación, que es lo que suele ocurrir. Para evitarlo, en este libro hemos preferido centrarnos en los conceptos y mecanismos de las redes aunque su explicación pueda involucrar varias capas. Pensamos que esto es más adecuado para novatos aunque quizá sea menos formal. En todo caso, no vamos a ignorar los modelos de referencia; son muy útiles para ubicar cada componente y entender cómo se relaciona con el resto. Los mencionaremos con frecuencia, simplemente no los utilizaremos como hilo conductor.

Veamos un pequeño resumen de la estructura general del libro:

Shell (C2) y Python (C3)

introducen dos tecnologías que vas a necesitar para seguir los frecuentes comandos de consola y pequeños programas que se utilizan en todo el libro. La shell y Python son el soporte pragmático del texto, de modo que si no tienes experiencia previa con ellos, es muy conveniente que los leas y practiques los ejemplos que incluyen. Si consideras que tienes suficiente familiaridad con ellos, puedes omitir estos capítulos.

Internet (C4)

aborda algunos de los conceptos más básicos: red, interred o protocolo, y los modelos de referencia, incluido el citado modelo OSI.

Protocolos esenciales (C5)

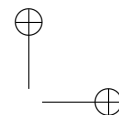
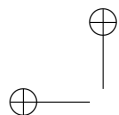
introduce los protocolos elementales de la familia TCP/IP. Aunque solo se cuenta lo mínimo de su funcionamiento, sí se presenta el formato de cada mensaje. Esto es así porque queremos que te pongas manos a la obra lo antes posible, que captures, veas, y entiendas tráfico real sin esperar a leer toda la teoría.

Sockets (C6)

Los *sockets* son la herramienta de programación elemental para escribir software de redes. Aunque muchos autores los consideran un tema avanzado y muchos libros los omiten completamente, aquí incluimos esta introducción muy al principio. Entender lo que el programador puede hacer o no te va a ayudar a comprender todo lo demás.

Direccionamiento IP (C7)

Las redes se utilizan para comunicar computadores —eso seguro que ya lo sabes— y para eso cada computador debe tener una dirección única. Este capítulo explica con detalle cómo se organizan las direcciones IP dentro de una red y cómo se puede saber a qué red está conectado cada computador a partir de su dirección.



Interconexión de redes (C8)

La interconexión de redes es el concepto clave de Internet. Aquí verás cómo los routers son capaces de mover paquetes de una red a otra, conceptos clave como la tabla de rutas, los métodos de entrega de paquetes o cómo se resuelve el problema de la fragmentación de paquetes.

Configuración IP (C10)

Aprenderás cuáles son los datos de configuración esenciales que requiere un nodo para poner conectarse de forma correcta a una red, cómo conseguir esos datos y cómo es posible automatizar el proceso.

Encaminamiento dinámico (C9)

Las redes cambian constantemente, y los routers necesitan adaptarse a esos cambios. Aquí verás los mecanismos que permiten a los routers trabajar juntos para encontrar la mejor ruta en cada momento y así llevar cada paquete hasta su destino.

Confiabilidad (C11) y Control de congestión (C13)

El protocolo de red IP no da ninguna garantía de que los datos vayan a llegar sin errores, ni siquiera que vayan a llegar. En estos dos capítulos verás cómo es posible añadir mecanismos que garanticen la entrega correcta de los mensajes, corrigiendo errores y retransmitiendo mensajes perdidos. También verás qué es la congestión de la red y cómo se combate.

Cliente-servidor (C14)

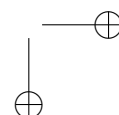
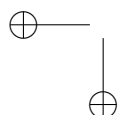
El modelo cliente-servidor es el que siguen la mayoría de aplicaciones de red. En este capítulo verás en qué consiste y cómo se pueden diseñar e implementar servidores eficientes y escalables aprovechando el soporte que ofrece la librería estándar de Python. También verás cómo aplicar técnicas de concurrencia, paralelismo y entrada/salida asíncrona.

Publicador-subscriptor (C15)

es otro de los modelos de interacción clásicos. Permite la comunicación entre múltiples participantes de forma más asíncrona y flexible que el modelo cliente-servidor, siempre claro, que el diseño de la aplicación lo permita. A pesar de no ser tan popular, es muy utilizado en aplicaciones de mensajería y propagación de notificaciones.

Serialización (C17)

trata sobre cómo las aplicaciones de red pueden intercambiar datos arbitrariamente complejos en formato binario o texto de forma efi-



ciente. Verás las bases de la codificación de datos aplicada a código listo para probar.

Captura y análisis (C18)

profundiza en el potencial que ofrecen las herramientas de captura de tráfico `tshark` y `wireshark`. El análisis del tráfico de red es importante para muchas situaciones y es clave en campos tan pujantes como la ciberseguridad. Aquí verás cómo capturar, filtrar y analizar tráfico de forma práctica y sencilla.

Sockets raw (C19)

explica qué son y para qué se utilizan este tipo de sockets y su papel en el tratamiento y generación de mensajes a bajo nivel. Verás también ejemplos prácticos para escribir programas capaces de capturar y generar mensajes de red a cualquier nivel de la pila de protocolos.

FiXme
Fatal!

FiXme Fatal: Completar resto de capítulos

Anglicismos

En el texto se utilizan muchos términos en inglés a pesar de que existen equivalentes en español. Esto es así porque priorizamos la claridad y la precisión técnica por encima de la corrección lingüística. Nuestro principal objetivo es que el lector entienda la redacción y comprenda los conceptos que se abordan, y eso no se consigue empleando términos que nadie utiliza en la ámbito profesional, por muy correctos que sean. Además, siempre que se introduzca un nuevo concepto, incluso si es de uso común en español, para evitar cualquier confusión se incluirá también el término en inglés entre paréntesis.

Hipertexto

En la versión pdf del libro, todas las referencias a capítulos, secciones (denotadas con §), figuras, tablas, listados de código, citas bibliográficas y acrónimos son enlaces que te llevan al lugar dónde se encuentran. Pero en la mayoría de los casos basta con colocar el cursor del ratón encima del enlace para que el visor pdf muestre una vista previa.

En cuanto a los acrónimos, como en cualquier documento técnico, aparecen en gran cantidad. Cuando se trate de un acrónimo poco habitual se definirá la primera vez que se utilice. Si es de uso más común tendrás que consultar el listado de acrónimos en la pagina XVII.

Ejemplos y código fuente

Todos los ejemplos que aparecen en el libro están disponibles para descarga a través del repositorio git en <https://github.com/destripando-internet/code>. Aunque es posible descargar este contenido individualmente o como un archivo comprimido, te aconsejamos utilizar el sistema de control de versiones git¹. Las rutas relativas a este repositorio que aparecen en el texto son ‘clickables’ y van precedidas del logotipo de github (🐙).

Si encuentras alguna errata u omisión es los programas de ejemplo, por favor, utiliza la herramienta de gestión de incidencias (*issue tracker*) accesible desde <https://github.com/destripando-internet/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 en <https://github.com/destripando-internet/book>. Si quieres colaborar activamente en su desarrollo o mejora, ponte en contacto con los autores.

Al igual que con los ejemplos, también existe una herramienta pública de gestión de incidencias en la que puedes notificar problemas o errores de cualquier tipo que hayas detectado en el documento.

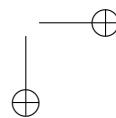
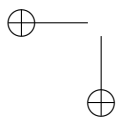
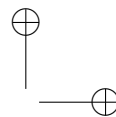
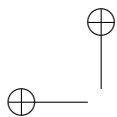
Exención de responsabilidad

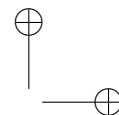
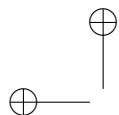
Este libro contiene abundantes ejemplos de comandos y programas destinados exclusivamente a fines formativos. En muchas ocasiones estos comandos implican privilegios de administración que pueden alterar la configuración y estado del sistema, y por tanto pueden tener consecuencias imprevistas y/o no deseables incluyendo la pérdida o exposición de datos o la interrupción de servicios.

Los autores y editores de este libro no son responsables de ningún daño o pérdida, directa o indirecta, como consecuencia de la utilización de los ejemplos o la información expuesta. Se recomienda al lector ejecutar siempre los ejemplos y pruebas en un entorno de laboratorio y nunca en sistemas en producción.

El lector asume toda responsabilidad por el uso que realice de los contenidos del libro.

¹<https://git-scm.com>





Capítulo 1

Debian GNU/Linux

En este capítulo —nada técnico— veremos de dónde viene el sistema operativo Debian, su filosofía y características. También veremos algo de la historia de sus precursores: el proyecto GNU, el núcleo Linux y el sistema UNIX. Aunque es una historia conocida, nos parece interesante incluir aquí un pequeño resumen de lo que es una parte crucial de la historia de la informática y que ayuda a entender cómo hemos llegado hasta aquí.

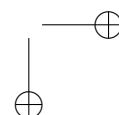
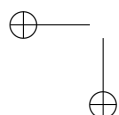
El relato nos lleva directamente a los albores de la informática y de la incipiente industria del software. Empecemos.

1.1. UNIX

Corría el año 1969, cuando un pequeño equipo de investigadores de Bell Labs (AT&T), liderado por Ken Thompson y Dennis Ritchie, empezó a trabajar en un nuevo sistema operativo inspirado en el proyecto Multics, en el que Bell Labs había participado, pero abandonó. Como un juego de palabras, llamaron UNIX al nuevo sistema ya que pretendía ser más sencillo que Multics.

El primer prototipo de UNIX, como era lo habitual entonces, se desarrolló en ensamblador. Para facilitar la experimentación, que era una parte fundamental de la labor de Bell Labs, decidieron reescribir UNIX en un nuevo lenguaje de programación que el mismo equipo estaba creando: el lenguaje C [2]. Desde entonces y hasta la actualidad, más de 50 años después, los SO se siguen escribiendo principalmente en C.

La decisión de usar C le proporcionó a UNIX un nivel de portabilidad sin precedentes, y eso a pesar de que no era el objetivo principal de ese cambio. Eso facilitó que fuera adaptado con rapidez a diferentes arquitecturas por parte de Universidades y otros centros de investigación, lo que resultó clave en su gran éxito y popularidad.



También fue determinante un conjunto de pautas bien definidas, conocidas como *filosofía Unix*. Algunos de sus principios más importantes son:

- Cada programa hace solo una cosa y la hace bien.
- Los programas trabajan con flujos. La salida de un programa es la entrada de otro.
- El texto plano es un formato de intercambio universal.
- Las tareas repetitivas son fáciles de automatizar (*scripting*).
- Minimiza la interacción con el usuario.
- Todo se trata como un fichero, con las mismas primitivas.
- Falla rápido e informa claramente del error.

Los distintos Unix¹ que surgieron dominaron la informática profesional desde entonces hasta comienzos del siglo XXI. De entre los muchos Unix que surgieron como AIX, HP-UX, SunOS, etc., uno de los más influyentes fue BSD, creado por la Universidad de California en Berkeley en 1977. BSD añadió funcionalidades muy importantes como el soporte para los protocolos TCP/IP o la abstracción de socket, y herramientas muy populares como el editor vi o la C Shell (csh). Además, su licencia mucho más permisiva que la del UNIX de AT&T, permitió que tanto su diseño arquitectural como su código fuera reutilizado incluso en productos comerciales privativos. De BSD surgieron variantes como FreeBSD, OpenBSD y NetBSD que siguen existiendo a día de hoy, y su influencia llega a otros SO de uso doméstico que surgieron después como Apple macOS o Microsoft Windows.

1.2. El fin del software libre

Ya antes de UNIX, empresas como IBM vendían y mantenían los grandes y caros computadores que se utilizaban en facultades e instituciones de investigación. Como parte de sus servicios, proporcionaban a los clientes el código fuente de los programas que se ejecutaban en sus computadores² como requisito para su funcionamiento. Los programadores e investigadores podían estudiar, modificar y mejorar esos programas. Los propios fabricantes fomentaban esa práctica, redistribuyendo esas modificaciones al resto de sus clientes, bajo la convicción de que el valor de su negocio estaba únicamente en el hardware.

¹«UNIX» es la marca comercial propiedad de AT&T —hoy un estándar de Open Group— mientras que «Unix» es un término genérico para referirse a los sistemas de este tipo.

²Como hacía AT&T con UNIX.

En 1976, un estudiante de 20 años del Harvard College, escribió una breve carta abierta titulada «Open Letter to Hobbyists»³ [3] en la que argumentaba que los programadores que estaban compartiendo sus aplicaciones y modificaciones perjudicaban a las empresas que comercializaban software, siendo según él, una actividad equivalente al robo. Abogaba por cerrar el código, distribuir únicamente binarios ejecutables, licenciar el uso y prohibir legalmente la modificación y redistribución.

La carta no tuvo mucha repercusión en un primer momento, pero el año siguiente (1977) el periodista Andrew Pollack publicó un artículo en «The New York Times» apoyando abiertamente estas ideas. Su difusión en un periódico tan influyente marcó un punto de inflexión que propició que rápidamente empresas como Apple, MITS, Tandy o Commodore comenzaran a cerrar su software. Posteriormente esos cambios llegaron también a la legislación de protección industrial e intelectual. El nombre del estudiante autor de la carta era William Henry Gates III, mas conocido como Bill Gates.

1.3. GNU

Una de las instituciones que empezaron a notar el cambio de mentalidad de la industria fue el Laboratorio de Inteligencia Artificial del MIT. Uno de los investigadores que trabajaba allí, Richard Stallman, vio su forma de trabajar amenazada por los recientes cambios en la legislación y en el modo de operar de las empresas de software. Stallman decidió iniciar el proyecto GNU [4] con el objetivo de crear un SO completo que permitiera a los usuarios recuperar la libertad perdida: la de usar, estudiar, modificar y distribuir el software sin restricciones.

El nombre GNU es un acrónimo recursivo. Significa GNU is Not Unix (GNU no es Unix) a la vez que juega con la pronunciación de la palabra *new*, que es prácticamente igual. El nombre plasma la intención de crear un SO similar a Unix, pero completamente libre, con el objetivo de que nadie tuviera ni quisiera usar ningún Unix propietario, terminando por reemplazarlo completamente.

Así nació la idea del software libre, aunque en realidad no era más que un intento por recuperar lo que algunos entendían que era la forma original y natural de entender el software. La decisión de crear un sistema similar a Unix no fue tanto por su superioridad técnica, sino porque era el SO

³*hobbyist* se refiere al ‘ecosistema hobbyist’, una comunidad de aficionados cuyo máximo exponente fue el Homebrew Computer Club (California, 1975).

más popular en ese momento, y Stallman entendió que eso ayudaría a su adopción.

Stallman y su equipo crearon la FSF para coordinar y promover el desarrollo del sistema GNU, y desarrollaron la licencia GPL, que constituye la base legal para proteger la libertad del software.

Con los años, el proyecto GNU fue creciendo y creando todo tipo de herramientas importantes como editores, compiladores, librerías, etc., A finales de los 80, el sistema tenía todos los componentes necesarios salvo uno: el núcleo. Trabajaban activamente en un núcleo llamado HURD, pero su diseño era muy ambicioso y complejo, y su desarrollo se retrasaba constantemente.

1.4. Linux

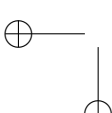
Linus Torvalds, otro estudiante, este de la Universidad de Helsinki, estaba estudiando Minix, un pequeño sistema operativo con fines educativos basado en la filosofía de Unix creado por Andrew Tanenbaum. Aunque Torvalds admiraba Minix como herramienta didáctica, encontraba limitaciones en su diseño y era crítico con ciertas decisiones que afectaban a su rendimiento. Frustrado por la limitaciones que Tanenbaum imponía a la modificación y evolución de Minix, en 1991 anunció a la comunidad de Minix [5] que estaba trabajando en Linux, su propio núcleo.

Pronto Torvalds decidió liberar Linux bajo la licencia GPL, si bien sus motivaciones siempre fueron más técnicas y pragmáticas que filosóficas o éticas. La FSF y el proyecto GNU adoptaron el núcleo Linux como una solución temporal hasta tener lista una versión estable de HURD, que nunca llegó. FSF desarrolló la librería glibc, un gran hito poco valorado, que permite que el sistema GNU pueda ejecutarse sobre el núcleo Linux, y también adaptaron muchas otras herramientas.

1.5. GNU/Linux

La combinación del sistema GNU y el núcleo Linux dio lugar a lo que conocemos como GNU/Linux, un SO completo y libre. Sin embargo, instalar un sistema así no era nada sencillo y solo personas con conocimientos técnicos avanzados podían conseguirlo. Implicaba compilar el núcleo y la mayoría de las herramientas.

Para simplificar esta tarea aparecieron las primera «distribuciones» de GNU/Linux. Una distribución es una colección organizada de paquetes, pre-compilados y mantenidos por un equipo de desarrolladores con el objetivo



de conseguir una coherencia entre ellos. Una distribución suele incluir el núcleo Linux —normalmente compilada con un objetivo específico—, el sistema GNU, un programa de instalación y una selección de programas y herramientas. Conforme todos esos componentes van evolucionando y se ofrecen versiones nuevas, la distribución define también nuevas versiones (*releases*). De ese modo los usuarios pueden instalar y mantener un sistema funcional con ciertas garantías en cuanto a estabilidad.

A lo largo de la década de los 90, GNU/Linux fue ganando popularidad y para principios de los 2000 era ya una alternativa viable a nivel industrial. Rápidamente se convirtió en la plataforma dominante para servidores en Internet, al mismo tiempo que copaba el Top 500 de los supercomputadores a nivel mundial, un nicho tradicionalmente dominado por los Unix comerciales. En 2010, con la compra de Sun Microsystems por parte de Oracle, Solaris que era de facto el último Unix propietario relevante, prácticamente desaparece.

1.6. Debian GNU/Linux

En 1993, otro estudiante de la Universidad de Texas, Ian Murdock, anunció el proyecto Debian [6]. Su objetivo era crear una nueva distribución de GNU/Linux que hiciera énfasis en la libertad.

Debian fue una de las primeras grandes distribuciones de GNU/Linux⁴, y se estableció rápidamente como una de las más profesionales y respetadas por la comunidad hasta la actualidad. En su compromiso por la libertad, Debian diferencia claramente entre software libre y no libre, de modo que los usuarios puedan tomar decisiones informadas sobre lo que instalan.

Uno de los aspectos distintivos de Debian es su Contrato Social⁵, un documento que establece los principios éticos y los compromisos que el proyecto asume con sus usuarios. Los podemos resumir en sus epígrafes:

1. Debian permanecerá 100 % libre.
2. Contribuiremos con la comunidad de software libre.
3. No ocultaremos los problemas.
4. Nuestra prioridad son nuestros usuarios el software libre.
5. Asumimos que algunos usuarios necesitan software no libre.

También define las Directrices de Software Libre de Debian (DFSG) que establecen los criterios que deben cumplir los programas que se incluyen en la distribución:

⁴Solo por detrás de Slackware

⁵https://www.debian.org/social_contract.es.html

1. Libre redistribución.
2. Disponibilidad del código fuente.
3. Permiso par realizar trabajos derivados.
4. Integridad del código fuente del autor.
5. No discriminación contra personas o grupos.
6. No discriminación en función de la finalidad.
7. Distribución de la licencia.
8. La licencia no debe ser específica para Debian.
9. La licencia no debe contaminar otro software.

Aunque su esfuerzo por mantener y promover la libertad del software es ciertamente destacable, no es el único aspecto que hace de este un proyecto de referencia entre las distribuciones de GNU/Linux. En el plano organizativo, Debian se basa principalmente en su comunidad de *desarrolladores oficiales*⁶, en su mayoría voluntarios. Convertirse en desarrollador oficial está abierto a cualquier persona, pero requiere superar un proceso de selección bastante riguroso.

Existe un conjunto de normas públicas que regulan todos los procesos importantes del proyecto, como pueden ser la adopción o eliminación de paquetes oficiales, cambios en la *policy*⁷ o la elección del líder. Todas estas decisiones se toman mediante votaciones públicas con debates en los que participa la comunidad. La ausencia de una empresa tras el proyecto y la gran transparencia del proceso son aspectos muy valorados y muy poco habituales en la industria del software, incluso en la del software libre.

Debian hace mucho énfasis en la estabilidad y el soporte a largo plazo. Para ir incorporando nuevas versiones de los programas, mantiene un ciclo de desarrollo basado en tres ramas diferentes: *stable*, *testing* y *unstable*⁸. La rama *stable* corresponde con las versiones oficiales (*releases*) y está pensada para usar en servidores y entorno de producción. Las versiones estables se publican aproximadamente cada 2 años y tienen un soporte de unos 3 años⁹. La rama *testing* es el entorno de pruebas para preparar la siguiente versión estable y, aunque puede seguir incorporando nuevas versiones de los paquetes, debe estar justificado. La rama *unstable* es la versión de desarrollo, y en ella pueden aparecer versiones nuevas prácticamente a diario.

Aunque es perfectamente posible utilizar *unstable* es un equipo de escritorio, existe corre un riesgo no despreciable de quedar con paquetes o dependencias rotas en momento puntuales. Esto permite a cada usuario decidir el

⁶Abreviado como DD (Debian Developer)

⁷Se refiere a la *Debian Policy Manual*: <https://www.debian.org/doc/debian-policy/>

⁸Y una mucho más inestable llamada *experimental*

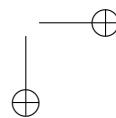
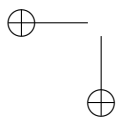
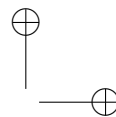
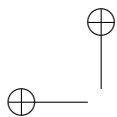
⁹Puedes ver la lista oficial de versiones estables en <https://www.debian.org/releases/>

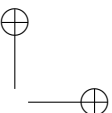
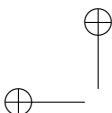
equilibrio de estabilidad/novedad que quiere para su sistema de una manera muy sencilla, y además el posible tener un sistema mixto con paquetes de diferentes ramas.

Debian, además, es una de las distribuciones que mantiene soporte para más arquitecturas. En el momento de escribir esto la última versión estable publicada (Debian 13: Trixie) cuenta con soporte para 9 arquitecturas. Esto da una idea clara de su compromiso para ofrecer al usuario un sistema operativo realmente universal.

A lo largo de sus muchos años de vida, Debian ha influido e influye en la industria del software. No solo por las muchas distribuciones derivadas¹⁰, algunas de ellas muy populares como Ubuntu, Kali o Raspbian, que siguen bastante integradas en sus procesos de desarrollo, tomando los paquetes actuales de Debian como base. También son referencia sus prácticas de desarrollo y empaquetado (especialmente su gestor `apt`), sus procesos de mantenimiento y despliegue, y su esquema de versiones y ciclo de desarrollo.

¹⁰Unas 120 según Distrowatch.com





Capítulo 2

Shell

Al terminar este capítulo, entenderás:

- Qué es una shell y por qué es tan importante.
- Cómo trabajar con procesos, trabajos y señales.
- Cómo redirigir la entrada y salida de los programas.
- Cómo gestionar usuarios, grupos, propietarios y permisos.
- Cómo instalar paquetes y configurar repositorios.
- Cómo gestionar servicios.
- Qué son los parámetros del núcleo y cómo configurarlos.

El intérprete de comandos (*shell*) es probablemente una de las herramientas más interesantes del sistema operativo GNU/Linux, y de cualquier POSIX en general. Una shell no es más que un programa interactivo que permite al usuario ejecutar otros programas manteniendo cierto control sobre ellos. Como con cualquier tecnología o herramienta que merezca la pena, es fácil encontrar grandes partidarios y detractores. Lo cierto es que para un novato la shell tiene un apariencia extraña, rudimentaria, obsoleta y de aspecto absolutamente espartano en un mundo en el que las pantallas multi-táctiles, la realidad virtual o el reconocimiento de voz son algo cotidiano.

Una interfaz de comandos —a menudo denominada 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 aumentan, 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 3D, herramientas CAD o de edición de imágenes o vídeo, con literalmente miles de opciones repartidas en innumerables menús. De hecho, no es extraño que algunos de estos programas con interfaces gráficas

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

tan complejas ofrezcan también algún tipo de sistema de comandos para sus usuarios más avanzados, como es el caso de AutoCAD.

Otra buena razón por la que la interfaz de comandos puede resultar más productiva es porque te da la posibilidad de crear secuencias de comandos (*scripts*), permitiendo automatizar tareas repetitivas. Por ejemplo, cuando se quiere explicar una secuencia de acciones, incluso de complejidad media, usar comandos es mucho más sencillo que dar instrucciones para acceder a menús, botones y listas desplegables, algo que a menudo requiere grabar en vídeo la secuencia de acciones —los llamados *screencast*. En el caso de la shell, además existe todo un lenguaje de programación llamado *C-Shell* que dispone de variables, condicionales, bucles, funciones, etc., así que se puede ir mucho más lejos que una mera secuencia de comandos.

En todo caso, sería un error subestimar o considerar anticuada una herramienta simplemente por el hecho de basarse en una interfaz de comandos, y desde luego lo es en el caso de la shell de GNU/Linux.

A menudo, las aplicaciones bien diseñadas definen un conjunto de acciones en una librería, que pueden ser utilizadas indistintamente desde *front-ends*² gráficos, línea de comandos, o incluso desarrollando otras aplicaciones a medida.

script

Un script es un archivo de texto que contiene secuencias de comandos. Estos comandos son interpretados por otro programa que lee y procesa el archivo. Hablamos de *shell scripts* cuando el programa que los interpreta es una shell.

2.1. GNU Bash

Entre la gran variedad de shells que Los sistemas POSIX han conocido desde principios de los años 70, **Bash** es probablemente una de las más veteranas, conocidas y potentes. Aunque **bash** tiene características muy interesantes (como el autocompletado contextual), primero debemos abordar cuestiones básicas que todo usuario avanzado debería conocer.

Además de la propia shell, hay una colección de programas (agrupados bajo 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.

²Se 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)

Estos y otros programas diseñados por terceros, pero que respetan la misma filosofía, junto con C-Shell, proporcionan un ecosistema 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, incluso maliciosas. Y como la shell es la forma más habitual y flexible para administrar un sistema GNU/Linux, aprovecharemos este capítulo para aprender lo básico sobre gestión de procesos, programas, usuarios, archivos y servicios.

2.2. Valor de retorno

Los sistemas POSIX asumen que cuando un programa acaba devuelve un entero. Este valor es recogido por su proceso padre (que puede ser una shell) y ofrece información muy útil acerca del resultado de la ejecución.

Un programa que realiza su función satisfactoriamente debería devolver siempre un valor 0. Puede devolver un valor distinto para indicar que hubo un problema concreto, que su página de manual debería explicar.

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:

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

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

```
~$ 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
```

En este caso, la shell `bash` crea un proceso en el que ejecuta el programa `ls` con el argumento `/`, que representa la raíz del sistema de archivos. Cuando el programa termina, la shell recoge el valor de retorno. Ese valor se almacena en una *variable de entorno* llamada `?`³.

³Sí, el signo de interrogación

Puedes imprimir el valor de cualquier variable utilizando el comando `echo` y colocando el símbolo `$` delante del nombre de la variable. O sea que puedes ver el valor de retorno de un comando si inmediatamente después de terminar, ejecutas:

```
~$ echo $?
0
```

Si se produce un error, el programa retorna un código de error, independientemente de que el programa muestre un mensaje de error. Lo puedes comprobar en el siguiente ejemplo:

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

echo

Escribe la cadena de texto que se le indique como argumento, incluyendo las variables (identificadores precedidos del símbolo `$`).

La documentación de cada programa indica el significado de cada uno de los posibles valores de retorno. Si consultas la página de manual de `ls` (con `man ls`) verás algo como:

```
Exit status:
 0      if OK,
 1      if minor problems (e.g., cannot access subdirectory),
 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, por ejemplo si el SO (u otro programa) terminó («mató») el proceso. Por ejemplo, si el usuario pulsa `Control-C` mientras un programa se encuentra en ejecución bajo el control de una shell, esta le enviará la señal `SIGINT` (-2), y será ese el valor de retorno del programa.

2.3. Opciones

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

El formato largo (con `--`) es el adecuado cuando se quieren comandos auto-explicativos (como en este libro) o en un *script*. El formato corto es más conveniente cuando el usuario escribe comandos directamente en la consola, por el ahorro de tiempo, obviamente.

Para evitar sobrecargar las explicaciones sobre cada comando, en este capítulo vamos a obviar la utilidad precisa de cada opción. El lector puede consultar la página de manual del programa (`man comando`) o con la opción `--help` para conocer la función de cada opción.

2.4. Procesos

Un ‘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 muchas de las propiedades de su *padre*, tales como privilegios y descriptores de archivos abiertos. Para listar 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`. Cada proceso tiene un identificador numérico único asignado por el SO, llamado PID. Es el número que aparece en la primera columna del siguiente ejemplo:

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

La segunda columna, TTY (TeleTYpewriter), es el terminal al que está asociado; la tercera, (TIME), el tiempo de procesador otorgado al proceso hasta el momento; y por último (CMD), el comando tal como se lanzó.

El programa `ps` dispone de una amplia variedad de opciones. Por ejemplo, la opción `f` muestra la «genealogía» de los procesos. Fíjate que las opciones de `ps` no van precedidas de guion, como es habitual. Veamos qué ocurre al ejecutar `ps f`:

```
~$ ps f
  PID TTY          STAT TIME COMMAND
 4970 pts/2    Ss   0:00 bash
 5107 pts/2    T    0:01 \_ gedit
 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 \_ code shell.tex
```

También aparece una columna adicional (STAT) que indica el estado del proceso. El primer carácter: R (*running*), T (*stopped*) y S (*sleep*); el segundo: 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`, que efectivamente está pensada para terminar el proceso, pero se puede enviar cualquier otra señal que se especifique como argumento. Puedes ver la lista de todas las señales con `kill -l`. El siguiente comando envía la señal `SIGKILL` (9) al proceso con PID 5200:

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

Algunas señales pueden ser ignoradas o bien capturadas por el programa para realizar acciones especificadas por el programador⁴. Este es el caso de `SIGTERM`, que se podría considerar una solicitud amistosa de terminación. Otras (como `SIGKILL`, que también termina el proceso) no pueden ser ignoradas, para así garantizar que el SO siempre tenga el control.

2.4.1. Trabajos

La shell crea un nuevo proceso hijo⁵ para ejecutar cada comando que se le pide. El comportamiento normal de la shell es esperar a que ese proceso termine antes de permitir la introducción de un nuevo comando —fácil de comprobar con el programa `sleep`. Esto se llama ejecución en *primer plano* o *foreground*.

Sin embargo, en la ejecución de un programa puedes pedirle a la shell que no espere a su terminación, permitiendo incluso que el comando quede en ejecución indefinidamente. A eso se le llama ejecución en *segundo plano* o *background*» Para conseguir que la shell ejecute un comando en segundo plano basta con añadir el símbolo *ampersand* (&) al final de la línea de comando:

```
~$ gedit &
[1] 19777
~$
```

Como se aprecia en el listado anterior, la shell imprime una línea con dos números y luego queda disponible, a la espera de que se introduzca un nuevo comando. El primer número [entre corchetes] identifica el ‘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

⁴Ejecuta `man signal` para más detalles

⁵La propia shell también se ejecuta en un proceso, que puede haber sido creado a su vez por otra shell.

ejecutar múltiples trabajos en segundo plano. La forma más sencilla de ver cuáles son es el comando `jobs`:

```
~$ firefox &
[2] 20847
~$ jobs
[1]-  Running                gedit &
[2]+  Running                firefox &
```

Si ejecutas el comando `fg` (*foreground*), el último comando ejecutado en segundo plano (marcado con el símbolo `+`) pasará a primer plano, bloqueando la shell. Opcionalmente admite como argumento el identificador del trabajo que quieres llevar a primer plano.

```
~$ fg %1
gedit
```

Si la shell se encuentra bloqueada en espera de la finalización de un comando en primer plano, puedes pararlo (no cerrarlo) pulsando `Control-Z` (la shell lo representa con `^Z`). Partiendo de la situación anterior:

```
~$ fg %1
gedit
^Z
[1]+  Stopped                gedit
~$ jobs
[1]+  Stopped                gedit
[2]-  Running                firefox &
~$
```

Un trabajo **parado** puede volver a estado de ejecución en primer plano si ejecutas el comando `fg`, o a segundo plano si ejecutas `bg`.

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

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

```
~$ kill -SIGKILL %2
[4]+  Killed                firefox
```

2.4.2. 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 utilizar su nombre (con `pidof` o `pgrep`):

```
~$ pidof firefox
105894
```

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

```
~$ fuser 17500/tcp
17500/tcp:      10589
```

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

```
~$ pkill firefox
~$ 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.

2.5. Entrada, salida y salida de error

En los sistemas POSIX los programas interactúan con el SO únicamente por medio de archivos—o de abstracciones que ofrecen el mismo interfaz. 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 archivo.

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

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

Eso significa que cuando un programa trata de escribir algo (*p. ej.* con la función `puts()` en C o `System.out.println()` en Java) lo envía a 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’⁶). Esta asociación entre la consola y los descriptores de archivo estándar la determina precisamente la shell y, mediante las órdenes adecuadas, el usuario puede alterar esa

⁶Aunque ‘consola’ y ‘terminal’ no significan exactamente lo mismo.

asociación para que la entrada y la salida de un programa queden ligadas a otra cosa, como un archivo en disco o incluso otro programa.

La ejecución anterior del comando `ls` / ‘el listado de archivos del directorio raíz’, apareció en pantalla porque su salida estándar corresponde por defecto a la consola.

2.6. Redirección

Es posible alterar la salida estándar de cualquier programa para enviarla, por ejemplo, a un archivo. Simplemente debes escribir el símbolo `>` (mayor-que) seguido del nombre del archivo:

```
~$ ls -l / > /tmp/root-files
~$
```

Repito: la ‘redirección de salida’ consigue que **cualquier** programa pueda enviar su salida en un archivo (o donde la shell determine) sin que el creador de ese programa tenga que haber hecho absolutamente nada para soportarlo.

/tmp

El directorio `/tmp` se utiliza por las aplicaciones para archivos *temporales* en los que almacenar logs o datos de sesión. El contenido de este directorio se elimina al apagar el computador.

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 archivo o directorio. La segunda es que esta vez *nada* ha aparecido en la consola. La lista de los nombres de archivo ha sido almacenada en el archivo `/tmp/root-files`.

Puedes ver el contenido de ese archivo con el programa `cat`:

```
~$ 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 archivo nuevo. Si existe un archivo con el mismo nombre, su contenido se sobrescribe. Pero existe otro tipo de redirección de salida que añade el contenido *al final* del archivo especificado. Se indica con doble mayor-que (`>>`):

cat

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

```
~$ date > /tmp/now
~$ date >> /tmp/now
~$ cat /tmp/now
Mon Jul 15 12:05:47 CEST 2024
Mon Jul 15 12:05:49 CEST 2024
```

Volviendo al archivo `root-files`, veamos cómo se podrían filtrar sus líneas de modo que aparezcan únicamente las que contengan la cadena «Jun» (los archivos modificados en junio). Para eso puedes utilizar el comando `grep` del siguiente modo:

```
~$ 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
```

date

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

`grep`, como muchos otros programas, usa 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:

```
~$ 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 archivos 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 archivo en disco. Resulta irrelevante.

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

```
~$ grep Jun < /tmp/root-files > /tmp/Jun-files
```

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

```
~$ 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 archivos temporales (`root-files` y `Jun-files`), pero si lo único que te interesa es el resultado final, hay una forma más sencilla y rápida de conseguir lo mismo sin necesidad de crear esos archivos: las tuberías.

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 archivos del directorio raíz modificados en junio ordenados por tamaño sería:

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

Nótese que `grep` y `sort` no tienen nombres de archivo en sus argumentos y que, por tanto, están leyendo líneas desde sus respectivas entradas estándar.

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

```
~$ ls -l / | grep Jun | sort -n -k5 > /tmp/root-jun-files-sorted-by-size
```

También se puede hacer redirección de salida con la captura hacia una variable. Se utiliza la sintaxis `$(comando)`. Por ejemplo, si quieres almacenar el número de archivos del directorio raíz en una variable, puedes ejecutar:

```
~$ num_files=$(ls -l / | wc -l)
~$ echo $num_files
24
```

Otra forma muy potente aunque no tan habitual es la *sustitución de procesos* cuya sintaxis es `>(comando)` o `<(comando)`. Permite utilizar la salida o la entrada de un comando como si fuera un archivo. En realidad es así, aunque ocurre fuera de tu vista. La shell crea un archivo temporal en `/dev/fd/` que contiene la salida del programa. Por ejemplo, podemos comparar la lista de procesos ahora y dentro de 2 segundos con:

sort

Ordena las líneas de los archivos 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.

wc

Cuenta el número de líneas, palabras y caracteres de los archivos indicados como argumentos (o de su entrada estándar) y lo escribe en su salida estándar.

```
~$ diff <(ps aux) <(sleep 2; ps aux)
```

Es similar a la tubería, pero es más versátil porque permite utilizar la salida o la entrada de varios comandos en el mismo comando tal como ves en el ejemplo anterior.

diff

Compara el contenido de dos archivos línea a línea y escribe las diferencias en su salida estándar.

La salida de error también se puede redirigir. Por ejemplo, `cat` al igual que la mayoría de programas, imprime un mensaje por su salida de error si le das un archivo que no existe. Puedes redirigir a un archivo con `2>file`

```
~$ cat no-existe
cat: no-existe: No existe el \file o el directorio
~$ cat no-existe 2> /tmp/cat-error
~$ cat /tmp/cat-error
cat: no-existe: No existe el \file o el directorio
```

Y por último, puedes redirigir un descriptor a otro, por ejemplo, la salida de error a la salida estándar. Mira este ejemplo:

```
1 ~$ echo hola > si-existe
2 ~$ cat si-existe no-existe
3 hola
4 cat: no-existe: No existe el \file o el directorio
5 ~$ cat si-existe no-existe > /dev/null
6 cat: no-existe: No existe el \file o el directorio
7 ~$ cat si-existe no-existe 2> /dev/null
8 hola
9 ~$ cat si-existe no-existe > salida 2>&1
10 ~$ cat salida
11 hola
12 cat: no-existe: No existe el \file o el directorio
```

El comando `echo` crea el archivo `si-existe` con el texto «hola». Después le pedimos a `cat` que lo imprima junto a otro que archivo que no existe. Por eso aparece el «hola» en la **línea 3** (que envía a `stdout`) y el mensaje de error en la **línea 4** (que envía a `stderr`). Vemos las dos porque por defecto ambas salidas están conectadas a la consola. En la **línea 5** redirigimos `stdout` a `/dev/null` (que la descarta), y por eso solo vemos el mensaje de error. En la **línea 7** es `stderr` la que descartamos, así que vemos solo el «hola». Por último en la **línea 9** redirigimos `stderr` a `stdout` y redirigimos esta última a un archivo `salida`, y vemos que al imprimirlo contiene las dos líneas. Aunque es contraintuitivo, la redirección `2>&1` debe ir al final al comando porque la shell evalúa las redirecciones de derecha a izquierda.

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 eficiente⁷.

2.7. Usuarios y grupos

Todo lo que ocurre en un sistema GNU/Linux depende de un usuario, que puede ser un humano o un servicio. El usuario se identifica con un número: el UID (User IDentifier). También hay grupos de usuarios, que también se identifican con un número: el GID (Group IDentifier). Cada usuario puede pertenecer a múltiples grupos y, para cada usuario, existe siempre un grupo que suele tener el mismo número.

Para simplificar el manejo de usuarios y grupos, se establece una correspondencia entre identificadores numéricos y nombres, que está almacenada en el archivo `/etc/passwd`. Cada fila de este archivo contiene el nombre del usuario, su UID, su GID, su nombre completo, su directorio personal (su *home*) y la shell que se ejecutará cuando el usuario inicie sesión. Aquí puedes ver algunas líneas de un archivo `/etc/passwd`:

```
root:x:0:0:root:/root:/bin/bash
backup:x:34:34:backup:/var/backups:/usr/sbin/nologin
alice:x:1000:1000:Alice Doe:/home/alice:/bin/bash
```

El usuario `root` es el superusuario del sistema, que tiene acceso a todos los archivos y puede ejecutar cualquier comando. Siempre tiene el UID 0. El usuario `backup` es un usuario del sistema que no tiene acceso a la consola (su shell es `nologin`). El usuario `alice` es el primer usuario normal, con su propio directorio personal (`/home/alice`) y `bash` como shell.

El archivo `/etc/group` contiene la correspondencia entre los nombres de grupo, su GID y los usuarios que pertenecen a cada uno. Aquí tienes algunas líneas de un archivo `/etc/group`:

```
root:x:0:
backup:x:34:
audio:x:29:alice,pulse
alice:x:1000:
```

Puedes ver el identificador de tu usuario y los grupos a los que pertenece con el comando `id`:

⁷La web <https://www.commandlinefu.com/> es una buena prueba de la gran versatilidad de la redirección y las capacidades de la shell.

```
~$ id
uid=1000(alice) gid=1000(alice)
↪ grupos=1000(alice),29(audio),44(video),46(plugdev),111(bluetooth)
```

Los identificadores de usuario son números a partir de 1000, mientras que los servicios y grupos del sistema tienen valores menores. Como puedes ver en el ejemplo, el grupo *audio* tiene el GID 29, y por cierto, identifica a los usuarios que pueden utilizar los dispositivos de sonido.

2.7.1. Propietarios y permisos

Todos los procesos son propiedad de algún usuario y ese usuario determina qué puede hacer el proceso, y a qué archivos y dispositivos puede acceder. Puedes ver el propietario de un proceso con la opción *u* del comando *ps*. En concreto el siguiente comando muestra todos los procesos del sistema y el propietario en la primera columna. Aquí solo incluimos algunas líneas para ilustrar el resultado. Si lo ejecutas en tu sistema, probablemente verás más de 200.

```
~$ ps aux
USER      PID %CPU %MEM    TTY STAT START   TIME COMMAND
root         1  0.0  0.0 ?        Ss   16:55   0:01 /sbin/init splash
root         2  0.0  0.0 ?        S    16:55   0:00 [kthreadd]
alice    5021  0.0  0.0 ?        Ss   16:55   0:01 /lib/systemd/systemd --user
alice    8031  1.4  2.6 tty2    Sll+  16:56   3:44 /opt/google/chrome/chrome
alice   18950  0.0  0.4 ?        Ssl   17:31   0:07 /usr/libexec/gnome-terminal-server
alice   33354  0.4  1.3 ?        Ssl   18:33   0:38 /usr/share/code/code .
alice  196069  0.0  0.0 pts/5    Ss   21:05   0:00 bash
alice  202403  0.0  0.0 pts/5    R+   21:11   0:00 ps aux
```

Los archivos (y eso incluye los dispositivos representados como tales) también son propiedad de algún usuario y de un grupo. Ya has visto cómo listar los propietarios de archivos con el comando *ls -l*.

```
~$ ls -l /
total 98
lrwxrwxrwx  1 root root      7 sep 27  2023 bin -> usr/bin
drwxr-xr-x  3 root root    4096 may  2 18:43 boot
drwxr-xr-x 19 root root    4100 may  8 21:14 dev
drwxr-xr-x 207 root root   16384 may  8 21:14 etc
drwxr-xr-x  6 root root    4096 dic  1 2023 home
drwx----- 22 root root    4096 may  4 15:00 root
-rw-r--r--  1 root root    1028 ene 19 2023 PKGBUILD
-rwx-----  1 root root      27 feb 16 2024 vmlinuz
[...]
```

La tercera y cuarta columnas indican el usuario y el grupo propietario. En el ejemplo todos son del usuario *root* porque obviamente son archivos del sistema. La primera columna tiene información interesante codificada con una serie de letras y guiones:

La primera letra indica el tipo de archivo: `d` para directorios, `l` para enlaces simbólicos y `-` para archivos normales. Después aparecen 9 letras que tienes que interpretar como tres grupos. Expresan los permisos de lectura (`r`), escritura (`w`) y ejecución (`x`) del usuario, el grupo y del resto de usuarios (tres letras para cada uno). Si no tiene el permiso, aparece un guion (`-`).

Por ejemplo, la línea `drwxr-xr-x` de `boot` significa que es un directorio (`d`), que el propietario puede leer, escribir y ejecutar (`rw`) el archivo y que tanto los miembros del grupo como el resto de usuarios solo pueden leer y ejecutar (`r-x`). Como es un directorio, ‘leer’ significa que se puede listar su contenido, ‘escribir’ que se puede crear o borrar archivos dentro de él y ‘ejecutar’ que puede acceder a su contenido.

Cuando se aplica a un archivo, ‘ejecutar’ significa que lo puede ejecutar como un programa. Por ejemplo, el archivo `vmlinux` es un archivo normal (`-`) que solo puede ser leído, escrito (modificado o borrado) y ejecutado con el usuario `root`.

Cuando un usuario ejecuta un programa, el SO crea un proceso que hereda el propietario que lo ha ejecutado y el SO se encarga de que las operaciones que pueda realizar ese proceso respeten los permisos de los archivos a los que trate de acceder.

Por ejemplo, el archivo `/etc/shadow` almacena información sensible relacionada con las contraseñas de cada usuarios. Por ese motivo, solo el usuario `root` y el grupo `shadow` tienen permiso de lectura.

```
~$ ls -l /etc/shadow
-rw-r----- 1 root shadow 2134 mar  3 12:24 /etc/shadow
```

Si intentas leerlo con un usuario normal, el SO te lo impedirá:

```
~$ cat /etc/shadow
cat: /etc/shadow: Permiso denegado
```



Aunque seas el administrador de tu propio PC, una buena práctica de seguridad es evitar utilizar el usuario `root` a menos que sea estrictamente necesario. Si ejecutas un programa malicioso como `root` le estarás dando acceso a todo el sistema, con lo que el daño que puede provocar es mucho mayor.

2.8. Paquetes

Muchas distribuciones de GNU/Linux utilizan *paquetes* para distribuir sus programas. Un paquete no es más que una colección de archivos estructurados en directorios para ser colocados en sus rutas adecuadas dentro del sistema de archivos cuando se instale.

En Debian GNU/Linux, Ubuntu u otras distribuciones derivadas, estos paquetes son archivos comprimidos con extensión `.deb`. Si descargas o dispones de uno de estos archivos, puedes ver su contenido con `dpkg`:

```
~$ dpkg -c gedit_48.1-9_amd64.deb
drwxr-xr-x root/root      0 2025-05-03 05:18 ./
drwxr-xr-x root/root      0 2025-05-03 05:18 ./usr/
drwxr-xr-x root/root      0 2025-05-03 05:18 ./usr/bin/
-rwxr-xr-x root/root 14568 2025-05-03 05:18 ./usr/bin/gedit
drwxr-xr-x root/root      0 2025-05-03 05:18 ./usr/lib/
[...]
```

Ahí puedes ver la ruta donde se va a instalar cada fichero contenido en el paquete. Son rutas relativas a la raíz del sistema de archivos. En este caso, el programa `gedit` se instalaría en `/usr/bin/gedit`.

Por supuesto, también puedes instalar el paquete:

```
~$ sudo dpkg -i gedit_48.1-9_amd64.deb
(Leyendo la base de datos ... 803908 ficheros o directorios instalados actualmente.)
Preparando para desempaquetar .../gedit_48.1-9_amd64.deb ...
Desempaquetando gedit (48.1-9) sobre (48.1-9) ...
Configurando gedit (48.1-9) ...
```

Sin embargo, una característica muy importante de estos paquetes es que tienen especificadas sus *dependencias*, es decir, otros paquetes que deben estar instalados previamente. Puedes comprobar esas dependencias (y otra mucha información) sobre un paquete con `apt-cache`:

```
~$ apt-cache show gedit
Package: gedit
Version: 48.1-9
Installed-Size: 1319
Maintainer: Debian GNOME Maintainers <pkg-gnome-maintainers@lists.alioth.debian.org>
Architecture: amd64
Replaces: gedit-plugin-text-size (<< 48)
Depends: gedit-common (<< 49~), gedit-common (>= 48~), gsettings-desktop-schemas,
↳ iso-codes, gir1.2-gtk-3.0 (>= 3.22), gir1.2-gtksource-300, gir1.2-tepl-6 (>= 6.12),
↳ libc6 (>= 2.38), libcairo2 (>= 1.2.4), libgdk-pixbuf-2.0-0 (>= 2.22.0),
↳ libgedit-amtk-5-0
[...]
```

Como se puede ver, para muchos de estos paquetes se indica la versión necesaria. En ese ejemplo, puedes ver que se requiere una versión del paquete `libc6` que sea mayor o igual a la 2.38.

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

El programa `apt`⁸ puede determinar, automática y recursivamente, las dependencias de un paquete, descargarlos e instalarlos en el orden correcto.

```
~$ 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 libamtk-5-0 libamtk-5-common libtepl-5-0
Paquetes sugeridos:
  gedit-plugins
Se instalarán los siguientes paquetes NUEVOS:
  gedit gedit-common gir1.2-gtksource-4 libamtk-5-0 libamtk-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]
```

2.8.1. 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 los archivos `.deb` están accesibles públicamente. Por ejemplo, para Debian es <https://ftp.debian.org/debian/> y para Ubuntu <https://archive.ubuntu.com/ubuntu/>. De ambos 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 de la distribución)⁹. En el caso de Debian, estas *releases* tienen un número de versión y un nombre asociado. Por ejemplo Debian 13 tiene el nombre *Trixie*.

En el archivo `/etc/apt/sources.list` se especifican los repositorios de paquetes y las *releases* que se quiere usar. Aquí puedes ver un ejemplo de ese archivo:

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

Esta línea en concreto dice que es posible instalar paquetes oficiales (*main*), contribuciones de terceros (*contrib*) y software con licencias no libres (*non-free*), desde el repositorio de Debian para la versión *Trixie*.

⁸También existe `apt-get`, que es una versión de más bajo nivel más adecuada para scripts.

⁹En <https://www.debian.org/releases/> puedes encontrar las *releases* de Debian

Este archivo puede contener muchos repositorios, y también se pueden crear otros archivos 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 ejecuta:

```
~$ sudo apt update
```

Es necesario hacer esto regularmente¹⁰ porque el contenido de los repositorios cambia y se añaden nuevos paquetes o versiones.

En Debian hay siempre tres versiones activas:

stable

Es la última versión publicada y su contenido no debería cambiar. Corresponde con *Trixie* en el momento de escribir este texto.

testing

Contiene los paquetes que se están preparando para una futura versión estable y por tanto, cambia continuamente. En este momento se denomina *Forky* 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*, que en realidad es el acrónimo de *Still In Development* (aún en desarrollo).

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 archivo `/etc/apt/sources.list`:

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

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

```
~$ sudo apt upgrade
```

¹⁰Las distribuciones actuales tienen aplicaciones que lo hacen automáticamente.

Si incorporas repositorios de versiones nuevas y quieres actualizar tu sistema con ellos, tendrás que ejecutar:

```
~$ sudo apt dist-upgrade
```

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

```
~$ lsb_release -a
No LSB modules are available.
Distributor ID: ^I Debian
Description: ^I Debian GNU/Linux 12 (trixie)
Release: ^I 13
Codename: ^I trixie
```

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

```
~$ apt-cache policy gedit
gedit:
  Instalados: 44.2-1
  Candidato: 44.2-1
  Tabla de versión:
    48.1-4 500
      500 http://deb.debian.org/debian sid/main amd64 Packages
    48.1-3 650
      650 http://deb.debian.org/debian testing/main amd64 Packages
  *** 44.2-1 700
      700 http://deb.debian.org/debian stable/main amd64 Packages
    100 /var/lib/dpkg/status
```

Esto quiere decir que no podrás 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 sus dependencias pueden afectar a otros paquetes.

2.8.2. Cómo encontrar paquetes

Es habitual necesitar un programa o una librería, pero no saber en qué paquete se encuentra. Veamos algunas formas de localizar paquetes, pensadas sobre todo para instalaciones de escritorio.

command-not-found

Este paquete modifica el comportamiento de la shell, de modo que al ejecutar un programa que no está instalado, en lugar de mostrar un simple mensaje «comando no encontrado», indica en qué paquete se encuentra ese programa que tratas de ejecutar. Por ejemplo:

```
~$ filezilla
No se ha encontrado la orden «filezilla», pero se puede instalar con:
sudo apt install filezilla
```

apt-file

El programa **apt-file** busca en los índices de paquetes de los repositorios de tu sistema. Puedes indicar una ruta o parte de ella y muestra todos los paquetes que coincidan. En este caso sirva para cualquier fichero, no únicamente ejecutables. Es necesario que mantengas ese índice actualizado con **apt-file update**.

```
~$ apt-file find bin/filezilla
filezilla: /usr/bin/filezilla
```

dpkg -S

Si ya tienes instalado el programa, pero necesitas saber a qué paquete corresponde puedes ejecutar lo siguiente:

```
~$ which gedit
/usr/bin/gedit
~$ dpkg -S /usr/bin/gedit
gedit: /usr/bin/gedit
```

2.9. Servicios

Los servicios del sistema¹¹ son procesos en segundo plano bajo el control del SO. Arrancan automáticamente al iniciar el sistema 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.

Estos servicios son gestionados por un *gestor de servicios* que se encarga de arrancarlos, pararlos o reiniciarlos en función de su configuración o del estado del sistema. El sistema de gestión de servicios más frecuente en GNU/Linux en la actualidad es **systemd**. Aunque ofrece muchas funciones, veamos aquí los comandos más básicos. Obviamente solo el administrador puede ejecutarlos.

systemctl status <servicio>

muestra el estado del servicio.

systemctl stop/start/restart <servicio>

arranca el servicio.

¹¹A menudo llamados *daemons*, un término acuñado por ingenieros del MIT en los primeros años de UNIX.

systemctl enable/disable <servicio>

habilita o deshabilita el servicio para que arranque automáticamente o no al iniciar el sistema.

systemctl list-units

muestra el estado de todos los servicios disponibles.

Por ejemplo, puedes ver el estado del servicio `ssh` (que permite conexiones remotas al sistema) con:

```
~$ sudo systemctl status ssh
• ssh.service - OpenBSD Secure Shell server
   Loaded: loaded (/usr/lib/systemd/system/ssh.service; enabled; preset: enabled)
   Active: active (running) since Fri 2025-08-01 15:21:30 CEST; 1 day 7h ago
  Invocation: a081694ecbb7403190db862ff993b604
     Docs: man:sshd(8)
           man:sshd_config(5)
  Process: 965 ExecStartPre=/usr/sbin/sshd -t (code=exited, status=0/SUCCESS)
 Main PID: 1045 (sshd)
    Tasks: 1 (limit: 19048)
  Memory: 2.4M (peak: 3M)
     CPU: 25ms
  CGroup: /system.slice/ssh.service
          └─1045 "sshd: /usr/sbin/sshd -D [listener] 0 of 10-100 startups"

ago 01 15:21:29 maine systemd[1]: Starting ssh.service - OpenBSD Secure Shell server...
ago 01 15:21:30 maine sshd[1045]: Server listening on 0.0.0.0 port 22.
ago 01 15:21:30 maine sshd[1045]: Server listening on :: port 22.
ago 01 15:21:30 maine systemd[1]: Started ssh.service - OpenBSD Secure Shell server.
```

Ahí puedes ver que el servicio está habilitado (*enabled*) y activo desde hace 1 día y 7 horas.

En distribuciones GNU/Linux más antiguas se utiliza el sistema `SysVinit` que permitía hacer una gestión similar con el comando `service`. Aunque `SysVinit` está en desuso, `systemd` ofrece comandos compatibles en las distribuciones modernas:

service <servicio> status

muestra el estado del servicio.

service <servicio> stop/start/restart

para, arranca o reinicia el servicio.

service ---status-all

muestra el estado de todos los servicios instalados.

2.10. Parámetros del núcleo

Existen muchos parámetros de configuración del núcleo del SO que se pueden consultar y modificar en tiempo de ejecución. Esa modificación se aplica inmediatamente sin necesidad del reiniciar el sistema.

Estos parámetros o variables del sistema están disponibles directamente para el usuario en el directorio `/proc` aunque lo que contiene no son archivos al uso. Se trata de un sistema de archivos virtual (llamado *procfs*) que expone una gran cantidad de información del núcleo y de los procesos en ejecución.

En el directorio `/proc/sys` en particular, es donde puedes encontrar los parámetros de configuración, organizados en subdirectorios en función de su categoría. Por ejemplo, el directorio `/proc/sys/fs` contiene los parámetros del sistema de archivos y `/proc/sys/net` contiene configuración de la red.

Para ver el valor de uno de estos parámetros, puedes usar simplemente el comando `cat`. Por ejemplo, el parámetro `/proc/sys/fs/file-max` indica el número máximo de archivos que pueden estar abiertos al mismo tiempo.

```
~$ cat /proc/sys/fs/file-max
9223372036854775807
```



Como todos los parámetros están en `/proc/sys`, cuando se haga referencia a alguno de ellos a lo largo del libro, se indicará su ruta relativa, es decir, para referirnos a `/proc/sys/fs/file-max` hablaremos del ‘parámetro del núcleo `fs/file-max`’.

Para modificar el valor de un parámetro se requieren permisos de superusuario, y consiste en escribir el valor deseado en el archivo virtual, algo que puedes hacer con el comando `echo`. Por ejemplo, el parámetro `vm/laptop_mode` indica si el sistema está en «modo portátil». En este modo, el SO trata de ahorrar energía y optimizar el uso del disco duro. Para activarlo, basta con escribir un valor distinto de 0 en el archivo:

```
~# echo 1 > /proc/sys/vm/laptop_mode
~# cat /proc/sys/vm/laptop_mode
1
```

Existe un programa específico para manipular estos parámetros llamado `sysctl`. La siguiente consola muestra su uso para leer y modificar el parámetro `vm/laptop_mode` de forma equivalente a lo visto previamente:

```
~# sysctl vm.laptop_mode  
vm.laptop_mode = 0  
~# sysctl -w vm.laptop_mode=1  
vm.laptop_mode = 1
```

Sin embargo, estos cambios no son permanentes. Si se reinicia el sistema, el valor del parámetro (y por tanto el comportamiento correspondiente) volverá al valor por defecto. Para que estos cambios sean permanentes, deben añadirse en el archivo `/etc/sysctl.conf`. Para el parámetro anterior, la línea que se debe añadir es:

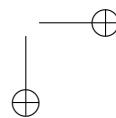
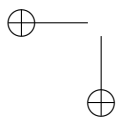
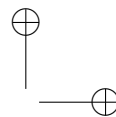
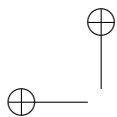
```
vm.laptop_mode = 1
```

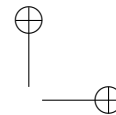
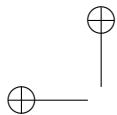
Y ¿qué más?

La shell representa como ningún otro programa la esencia de la *filosofía Unix*, que se puede resumir en una cita atribuida a Doug McIlroy, uno de los ingenieros de Bell Labs que diseñó el sistema UNIX:

Escribe programas que...
...hagan solo una cosa y la hagan bien.
...que trabajen juntos.
...que manejen flujos de texto, porque es una interfaz universal.

Aparte de los programas que hemos visto en este capítulo, existen muchos otros diseñados con estos mismos principios, que pueden ser combinados con redirección para realizar tareas complejas. El anexo A incluye una lista de comandos habituales, aunque a lo largo del libro también se introducirán algunos más.





Capítulo 3

Python

Al terminar este capítulo, entenderás:

- Cuáles son los tipos de datos básicos de Python.
- Cómo se definen funciones, clases y módulos.
- Cuáles son las estructuras de control.
- Cómo funciona la comprobación opcional de tipos.

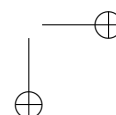
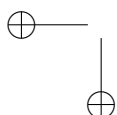
Python es un lenguaje interpretado, interactivo y orientado a objetos. Si ya conoces algún lenguaje como Java, la mayoría de lo que encontrarás en Python te sonará familiar. Tiene variables, clases, objetos, funciones y todo lo que se espera que tenga un lenguaje de programación «convencional». Cualquier programador puede empezar a trabajar con Python con poco esfuerzo. Según muchos expertos, Python es uno de los lenguajes más fáciles de aprender y que permiten al novato ser productivo en menos tiempo, y todo eso también explica que sea uno de los lenguajes más populares en la actualidad.

Python es perfecto como lenguaje «pegamento» y para prototipado. Dispone de librerías para desarrollar aplicaciones multihilo, distribuidas, bases de datos, con interfaz gráfica, juegos, gráficos 3D, cálculo científico, análisis de datos, inteligencia artificial y prácticamente todo lo que se te ocurra.

Este pequeño tutorial asume que dispones de Python versión 3.11 o posterior y tienes nociones básicas de programación con algún lenguaje imperativo u orientado a objetos.

3.1. ¡A programar!

Nada como programar para aprender un lenguaje, y en eso Python tiene una ventaja: el intérprete de Python se puede utilizar como una shell (y por eso se dice que es «interactivo»). Es algo tremendamente práctico para probar cosas, pero también para dar tus primeros pasos. Simplemente abre



una consola y ejecuta `python3`, o `python` dependiendo de la configuración de tu SO:

```
$ python3
Python 3.11.9 (main, Apr 10 2024, 13:16:36) [GCC 13.2.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

3.2. 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 porque viene con un «recolector de basura».

```
$ 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. Eso significa que no se puede operar alegremente con variables de tipos diferentes.

```
>>> 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` que avisa que no hay una forma implícita de convertir el `1` (un entero) para poder utilizar el operador `+` con `a` (una cadena). Las excepciones son el mecanismo más común de notificación de errores.

Sin embargo, puedes cambiar el tipo de la variable `a` sobre la marcha.

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

En realidad no has cambiado el tipo. Has creado una nueva variable que reemplaza a la anterior.

3.3. Tipos de datos

3.3.1. Valor nulo

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

3.3.2. Booleanos

Nada fuera de lo común:

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

3.3.3. Numéricos

Python dispone de tipos enteros de cualquier tamaño que soportan todas las operaciones aritméticas habituales, incluidas las de bits. También hay reales y ambos tipos pueden operar entre sí sin ningún 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 y permite operaciones aritméticas. En Python, sin embargo, se utilizan cadenas de caracteres de tamaño 1 y no permiten operaciones aritméticas.

3.3.4. Secuencias

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

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

El tipo **bytes** permite manejar secuencias de bytes. Se utilizan habitualmente para serialización de datos. Por ejemplo, puedes codificar una cadena de caracteres a UTF8 como secuencia de bytes y viceversa:

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

Cuando se utiliza codificación UTF8 algunos caracteres ocupan más de un byte. Este es el caso de las letras ‘ñ’ y ‘ú’ del ejemplo. Veremos esta cuestión con más detalle en el capítulo 17.

Las **tuplas** son una agrupación de valores similar al concepto matemático homónimo. Se pueden empaquetar y desempaquetar varias variables, y pueden ser de tipos diferentes.

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

La tupla, al igual que la cadena, es inmutable, 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 la quieres cambiar, tienes 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, con la diferencia de que 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 tipo de secuencia (listas, tuplas o 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 tables*). Tanto las claves como los valores pueden ser de cualquier tipo¹, incluso de tipos diferentes en el mismo diccionario:

```
>>> notas = {'antonio':6, 'maria':9.3, 'juan': 'No presentado'}
>>> notas['antonio']
6
```

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

3.4. 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
```

3.5. Estructuras de control

Están disponibles las más comunes: `for`, `while`, `if`, `else`, `break`, `continue`, `return`, etc. Todas funcionan de la forma esperable excepto `for`, que no es un `while` disfrazado. Este `for` itera sobre una secuencia de modo que la

¹En realidad tiene que ser un tipo *hashable*, es decir, que se pueda asimilar a un valor único

variable de control toma el valor de cada uno de sus elementos en cada iteración; en realidad es más parecido al `for_each` de otros lenguajes:

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

3.6. Indentación estricta

En programación se aconseja «tabular» (indentar) el código para alinearlos con los bloques y así ayudar a su lectura. En Python este estilo es obligatorio porque el esquema de indentación es lo que define los bloques. Es una importante peculiaridad de este lenguaje y obliga a utilizar un editor de código que controle correctamente esta cuestión. Se aconseja tabulación ‘blanda’ de 4 espacios. Mira el siguiente fragmento de código:

```
total = 0
passed = 0
grades = {'antonio':6, 'maria':9}
for grade in grades.values():
    total += grade
    if grade >= 5:
        passed += 1

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.

3.7. Funciones

Las funciones utilizan una sintaxis muy sencilla e intuitiva. Nada mejor para explicar su sintaxis que un ejemplo:

```
def factorial(n):
    if n == 0:
        return 1

    return n * factorial(n-1)

print(factorial(10))
```

3.8. Python is different

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

```
from functools import reduce
from operator import mul

factorial = lambda x: reduce(mul, range(1, x+1), 1)
print(factorial(10))
```

Si no conoces nada de programación funcional, sirva este ejemplo como mínima introducción. La palabra reservada `lambda` crea una función anónima, que por ejemplo se podría pasar directamente como argumento a otra función. En este caso se almacena en la variable `factorial`, de modo que la podremos usar del mismo modo que la función tradicional del ejemplo anterior. Esa función utiliza la `reduce()`, que es otra herramienta básica de programación funcional. La función `reduce()` aplica una función a los elementos de una secuencia. En este ejemplo, aplica el operador de multiplicación (`mul`) a los enteros del rango `1-x` usando un valor inicial de 1, es decir, multiplica $1 \cdot 1 \cdot 2 \cdot 3 \cdots x$.

3.9. Hacer un ‘ejecutable’

Lo habitual en programación es escribir el programa en un archivo de texto y luego compilarlo. Pero como Python es un lenguaje interpretado, no es necesario ningún tipo de procesamiento, al menos no explícito. Simplemente escribe el código en un archivo con extensión `.py`.

Para que la shell sepa qué programa debe ejecutar para interpretar el código de este archivo, es necesario añadir un comentario especial en la primera línea. Esta línea se conoce como «shebang» y tiene el siguiente aspecto:

```
#!/usr/bin/python3
```

Además debes darle permisos de ejecución al archivo:

```
$ chmod +x programa.py
```

Con eso ya puedes ejecutar el programa:

```
$ ./programa.py
```

Al encontrar el shebang, la shell ejecutará el programa como si hubieras escrito `/usr/bin/python3 programa.py`.

En el caso de Python, es más adecuado usar el programa `env` para localizar el intérprete adecuado de Python. El shebang queda así:

```
#!/usr/bin/env python3
```

3.10. Orientado a objetos

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

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

Escribir una clase, al menos algo básico, es muy sencillo:

```
class Book:
    def __init__(self, title, author):
        self.title = title
        self.author = author
        self.bookmarks = []

    def add_bookmark(self, page, label):
        self.bookmarks.append((page, label))

    def last_bookmark(self):
        return self.bookmarks[-1]

book = Book("1984", "George Orwell")
book.add_bookmark(101, "start of chapter 3")
print(book.last_bookmark())
```

Lo más importante es que el primer argumento de los métodos es siempre `self`, que representa la instancia de la clase sobre la que se ejecuta (`Book` en el caso del ejemplo). El método `__init__()` es el constructor² y se ejecuta al instanciar la clase: `book = Book("1984", "George Orwell")`.

3.11. *Type checking*

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

²En realidad es el *inicializador*, pero para un nivel básico nos sirve como idea

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

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

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

¿Y para qué sirve entonces poner el tipo? Principalmente para facilitar la comprensión del código, y sí, también para poder llevar a cabo comprobaciones de tipo (*type checking*), pero para eso se requieren herramientas adicionales como MyPy³.

Veamos el archivo `factorial.py` con una versión de la función de la §3.7, pero que incluye anotaciones en sus atributos y valor de retorno. También hemos añadido una llamada a la función que toma un argumento de la línea de comandos:

```
import sys

def factorial(n: int) -> int:
    if n == 0:
        return 1

    return n * factorial(n-1)

factorial(sys.argv[1])
```

Si ejecutas MyPy para que busque errores en esta implementación, verás 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 te avisa que el argumento que estás pasando a la función es una cadena, un tipo de dato que no se corresponde con el esperado (un entero). El error desaparece si transformas el parámetro de entrada del programa en un entero con `int(sys.argv[1])`.

Y ¿qué más?

Esto no ha sido más que una muy breve introducción. Como has podido comprobar Python es un lenguaje muy accesible, con una sintaxis sencilla. Pero eso no significa que sea un lenguaje simple. Aparte de la extensa librería estándar, hay disponible una ingente cantidad de módulos de terceros

³<http://mypy-lang.org/>

para casi cualquier necesidad imaginable. Indudablemente es un lenguaje que vale la pena aprender a dominar, con una gran comunidad de usuarios y excelente documentación.

Capítulo 4

Internet

Al terminar este capítulo, entenderás:

- Qué es una red de computadores.
- Qué es una interred.
- Para qué sirve un protocolo y qué es una pila de protocolos.
- Qué son y para qué sirven los modelos de referencia OSI, TCP/IP e híbrido.

Como ‘Internet’ es una palabra tan habitual, es normal pasar por alto su significado incluso entre personas con formación técnica, a pesar de ser bastante evidente. La palabra ‘internet’ (en minúscula) surgió como una abreviatura de **inter-network** (interred), que es un concepto amplio para designar cualquier sistema formado por la interconexión de múltiples redes de computadores. ‘Internet’ (empezando por mayúscula y sin artículo) es un nombre propio que designa específicamente a la más grande y compleja de las interredes, la que todos conocemos y usamos a diario¹.

Esto implica que cualquier otra colección de redes interconectadas, incluso sin usar tecnología TCP/IP, también es un ‘interred’. Como queremos que el lector tenga presente esta importante distinción, en el libro utilizaremos la palabra ‘interred’ siempre que la explicación se pueda aplicar a cualquier interred, y reservaremos la ‘Internet’ para la interred global.

También es interesante que sus creadores eligieran esa palabra. Significa que esa idea de interconectar redes es lo que les pareció su característica más destacable, al menos en ese momento.

¹Ten en cuenta que por su obvia similitud semántica existe abundante bibliografía, sobre todo en inglés, en la que «Internet» se utiliza como sinónimo de «interred», por ejemplo la RFC 1819 [7]. Estos documentos asumen que el lector interpreta el significado en función del contexto.

Una **red de computadores** es un conjunto de computadores autónomos² conectados entre sí de modo que puedan compartir recursos (datos o dispositivos). Todos los computadores de una misma red utilizan una misma tecnología de comunicaciones (*p. ej.* WiFi). Decimos que esto es «un enlace», o simplemente «una red». Y una red, incluso aislada del resto del mundo, puede tener utilidad, por ejemplo, para el control de una planta de fabricación industrial.

Existen dos tipos de enlace: enlaces compartidos y enlaces punto a punto. En un enlace compartido (como en una LAN) todos los computadores comparten un único medio físico (como el aire en el caso de WiFi), mientras que en un enlace punto a punto, el medio es utilizado únicamente por dos dispositivos (como un cable de cobre que los interconecta). Cada tecnología implica un tipo de enlace y responde a unas necesidades específicas. Veremos algunas de esas tecnologías más adelante.

En muchas ocasiones, la estructura de una red o interred es desconocida o irrelevante para el contexto específico que se está considerando. En estos casos se utiliza una nube como metáfora visual y lo único relevante es que hay ciertos dispositivos conectados a ella y gracias a eso pueden comunicarse.

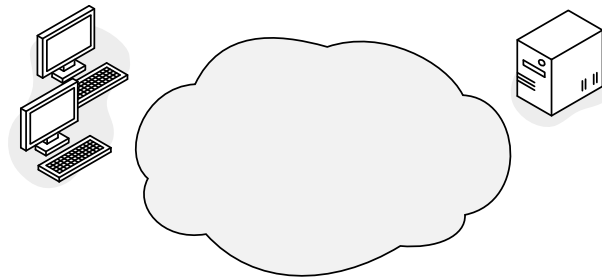


FIGURA 4.1: La nube como metáfora de una red o interred

4.1. Tecnología Internet

Como en tantos otros campos, no existe una única tecnología maravillosa que resuelve todos los problemas. Por ejemplo, no hay una tecnología universal para hacer redes locales o enlace punto a punto, hay decenas, y ninguna es perfecta (vaya, ¡menuda sorpresa!). La solución que propone la tecnología tras Internet es IP³ y no consiste en sustituir todas esas tecnolo-

²«autónomos» en el sentido de que pueden realizar tareas por si mismos

³Por supuesto no es el único modo, pero es el que abordamos en este texto.

gías sino integrarlas, interconectando redes heterogéneas –y probablemente incompatibles entre sí.

La tecnología TCP/IP logra esa interconexión gracias a tres elementos:

- El protocolo IP, que significa literalmente *inter-net protocol*, es decir, **protocolo de interred** implica un formato de mensaje universal (el **paquete** IP), que utilizan tanto los dispositivos terminales como los de interconexión (*routers*). IP no sustituye a los protocolo de enlace de cada red, sino que «se encapsula» en ellos⁴.
- El **direccionamiento** IP. Es un forma de asignar direcciones (números únicos) a cada dispositivo. Cualquier computador con capacidad de enviar y recibir paquetes IP–en cualquier parte– debe disponer de una dirección IP adecuada⁵.
- Dispositivos capaces de reenviar paquetes entre redes y hacerlos llegar a sus correspondientes destinos. Estamos hablando los routers o pasarelas IP (*gateways*). Y por supuesto se necesitan los mecanismos y protocolos que coordinan esos *routers*, que recibe el nombre de **encaminamiento**, encaminamiento o *routing* en inglés.

Internet es posible porque todos los dispositivos conectados se intercambian mensajes del protocolo IP, y eso junto con las direcciones IP y los routers, resuelve un problema clave: permite llevar paquetes con datos (sean lo que sean esos 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.

Estos 3 aspectos son probablemente los más importantes en el funcionamiento de Internet y los estudiaremos en detalle en el resto del libro.

4.2. Protocolos

Pero ¿qué es un **protocolo**? Sin meternos ahora en detalles, podemos decir que un protocolo es una serie de normas y reglas que deben cumplir los participantes para comunicarse de forma eficaz y productiva. Por ejemplo, para hablar con un policía hay una serie de expresiones y palabras que nunca se te ocurriría usar a menos que quieras meterte en problemas, e igualmente, el tipo de respuestas que cabría esperar también caen dentro de unas pautas previsibles. Y el modo en que hablas con un policía es diferente al que utilizas con un médico, un niño, un jefe, un compañero, etc. De

⁴Veremos más adelante qué es la encapsulación.

⁵Sí, todo se llama IP: protocolo IP, paquete IP y direcciones IP.

hecho, la palabra «protocolo» tiene acepciones que se aplican precisamente a las interacciones con personas en situaciones muy formales o solemnes, como las relaciones diplomáticas, el ámbito institucional o ciertas ceremonias oficiales. En esencia, la idea subyacente es la misma: establecer y cumplir pautas comunes para que la comunicación funcione.

Cuando dos dispositivos o procesos hablan entre sí a través de una red también deben cumplir una serie de normas y formatos; pero con mucha menos ambigüedad que entre personas. Los protocolos de red establecen una serie muy concreta de mensajes, con estructuras perfectamente especificadas que indican qué se puede enviar y responder, y cuándo está permitido enviar cada cosa. Por ejemplo, la comunicación entre un cliente y un servidor web utiliza exclusivamente el protocolo HTTP. Este protocolo solo admite una serie limitada de tipos de mensaje: GET, POST, HEAD, DELETE, etc. y cada uno de ellos tiene su estructura y formato. Si por cualquier motivo un navegador web envía un mensaje de tipo desconocido o que no respeta la estructura esperada, la comunicación acabará mal.

Los protocolos de bajo nivel (los de las capas inferiores) abordan problemas básicos generales como el transporte de mensajes, su división y re-ensamble, la detección o corrección de errores, etc., cuestiones que son comunes a cualquier tipo de comunicación, sin importar el contenido de los mensaje. Pero hay otros problemas que resolver.

El sistema operativo debe ofrecer una forma que permita que dos procesos ejecutándose en nodos cualesquiera puedan «direccionarse» el uno al otro, es decir, que puedan designar unívocamente a otro proceso que se ejecuta en un nodo remoto accesible a través de la red. El direccionamiento de procesos dentro de un nodo se denomina **multiplexación**. Esto se logra mediante la combinación de dos datos: una dirección IP, que identifica al nodo, y un puerto, que identifica un proceso dentro de ese nodo.

Además, muchas aplicaciones necesitan garantías de que estos mensajes lleguen correctamente a su destino, sin cambios (respetando su integridad), sin partes ausentes (omisiones), recibidos en el mismo orden en el que se enviaron, asegurando la privacidad, sin saturar al receptor, evitando congestionar la red, etc.

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

TCP

Proporciona un flujo (*stream*) ordenado y fiable de datos entre dos procesos.

UDP

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

SSL/TLS

Ofrece mecanismos para cifrado de mensajes y permite asegurar la legitimidad del proceso remoto.

Estos tres protocolos son genéricos porque se pueden aplicar sobre cualquier carga útil (*payload*), es decir, son independientes del contenido de los mensajes que las aplicaciones necesiten enviar, aunque eso no implica que sean intercambiables.

El formato y significado específico de los datos que utiliza cada aplicación también está definido por protocolos —llamados **protocolos de aplicación**. Cuando hablamos de «aplicación» no nos estamos refiriendo necesariamente a un programa concreto. Por ejemplo, todo el software de cualquier fabricante que está relacionado con la web utiliza el mismo protocolo de aplicación HTTP. Al contrario de lo que sucede con los protocolos de interred y transporte, hay miles de protocolos de aplicación, pues cada aplicación o servicio tiene su propio objetivo y necesidades de comunicación específicas.

Hay muchos protocolos de aplicación privativos y secretos. Sin embargo, hay unos cuantos que son públicos y abiertos, bien documentados y de uso muy común. Algunos de estos son DNS, SMTP/IMAP/POP, FTP, SSH o el ya mencionado HTTP. Estos protocolos abiertos permiten precisamente que cualquier persona o empresa pueda desarrollar una nueva aplicación que interaccione con las que existen, lo que fomenta competencia e innovación, y favorece el crecimiento de Internet.

4.3. Pila de protocolos

Para manejar más cómodamente los distintos problemas relacionados con la comunicación, su estudio, diseño, especificación e implementación de sus soluciones, los protocolos se organizan en capas o niveles formando una **pila** de protocolos (*protocol stack*). Se organiza así porque los protocolos de cada capa se relacionan con los de la capa justo por debajo y por encima de la suya. Todo esto es principalmente una conveniencia de **abstracción**, aunque no por ello es menos útil para entender cómo funciona el conjunto. Como el criterio para definir esta pila y qué poner en cada capa no es único, existen varios modelos de referencia. Los más importantes son el modelo OSI, el TCP/IP y el híbrido, que combina los otros dos.

4.4. 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.

Veamos brevemente el objetivo de cada capa empezando desde abajo:

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) dentro de una misma red. Cuando la red es un medio de difusión (como una LAN), suele proporcionar un sistema de *direccionamiento físico* ya que, si puede haber varios vecinos escuchando, es necesario indicar de algún modo a quién está realmente dirigido el mensaje.
3. **Red** – Proporciona soporte para comunicaciones *extremo a extremo*, es decir, que implica también dispositivos intermediarios. Permite enviar mensajes individuales de tamaño variable y define un sistema de *direccionamiento lógico*. Una de sus funciones más importantes es la capacidad de interconectar redes de tecnología distinta formando interredes.
4. **Transporte** – Proporciona un canal de comunicación libre de errores entre procesos remotos. Incluye un mecanismo de multiplexación y un sistema de direccionamiento de procesos.
5. **Sesión** – Permite crear sesiones entre nodos remotos y se ocupa de la sincronización.
6. **Presentación** – Define la representación canónica de los datos (reglas de codificación) y su semántica correspondiente.
7. **Aplicación** – Incluye los detalles y protocolos específicos de cada aplicación.

4.4.1. Direccionamiento físico vs. lógico

??

Los dos tipos de direccionamiento a menudo resultan confusos para los principiantes, pero es importante entender su utilidad y sobre todo saber diferenciarlos. Lo primero que debes saber es que cada tecnología está ligada a un tipo de direccionamiento.

Los protocolos de red/interred (como el protocolo IP) utilizan direccionamiento lógico. La dirección lógica (la dirección IP en el caso del protocolo IP) sirve para determinar un destino global, es decir, identifica un nodo que habitualmente está conectado a una red diferente, eventualmente en el extremo opuesto del planeta.

Por contra el direccionamiento físico es propio de los protocolos y tecnologías de enlace (como WiFi). La dirección física⁶ identifica el nodo al que entregar un mensaje en un ámbito local, es decir, el destinatario del mensaje es un nodo vecino, un nodo al que se puede llegar sin pasar a través de ningún intermediario. Por eso, las direcciones físicas no tienen sentido ni utilidad si el destino está fuera del enlace o red local.

En próximos capítulos veremos qué aspecto tienen las direcciones físicas y lógicas, y cómo se utilizan en la práctica. De momento, lo importante es recordar que una dirección física identifica un vecino, mientras que una lógica puede identificar cualquier nodo conectado a la interred, en cualquier parte. Y también recuerda que cada tecnología de comunicación utiliza uno u otro tipo de direccionamiento, pero no ambos.

4.5. Modelo TCP/IP

El modelo de referencia TCP/IP se definió años después de las primeras implementaciones de esos protocolos y trata de formalizar y estandarizar su uso para garantizar interoperabilidad entre los distintos fabricantes. Este modelo define únicamente cuatro capas, que vemos también de abajo hacia arriba:

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.

⁶También conocida como dirección hardware o MAC

2. **Interred** – Proporciona los mecanismos de interconexión de redes y encaminamiento de paquetes. Define el protocolo IP, que proporciona un servicio de entrega de tipo «datagrama», es decir, que lleva cada paquete hasta el destino de forma independiente, sin considerar si tiene relación con el anterior o el siguiente.
3. **Trasporte** – Proporciona mecanismos de comunicación entre procesos. Define en esencia 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 y todas las implicaciones funcionales que conlleva.

4.6. Modelo híbrido

Por último, el modelo híbrido es una mezcla de los modelos OSI y TCP/IP eliminando «lo que sobra» al modelo OSI (las capas de presentación y sesión son aplicables en relativamente pocas ocasiones) y añadiendo «lo que falta» al modelo TCP/IP. La capa de enlace es decisiva para comprender el funcionamiento de la arquitectura de red. La Figura 4.2 muestra la correspondencia entre los tres modelos descritos.

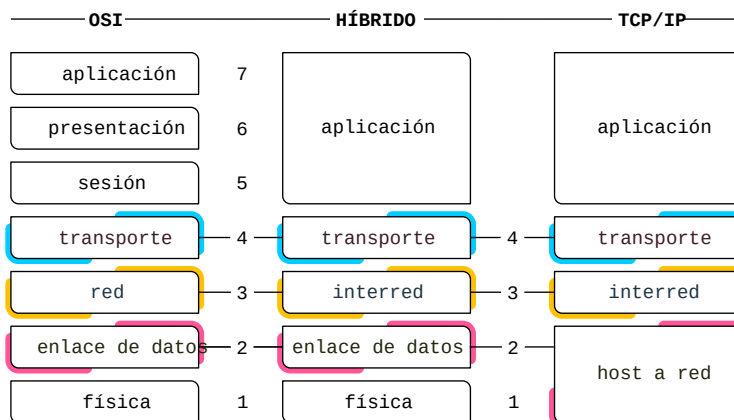


FIGURA 4.2: Correspondencia entre los modelos OSI, TCP/IP e híbrido

4.7. Qué no es Internet

El término «Internet» es de uso tan común que es inevitable que incluso usuarios con formación técnica acaben confundiendo otros conceptos más o menos relacionados con el que hemos visto en las secciones anteriores.

4.7.1. Internet no es la web

La web, o más precisamente la WWW (World Wide Web), es un servicio que, al menos inicialmente, proporcionaba un sistema de hipertexto que permite la navegación dentro de un documento (página) o entre varios. Obviamente esta sigue siendo una de sus funciones básicas, pero hoy en día es mucho más. La web se basa en un protocolo de aplicación (HTTP) y necesita de toda la infraestructura de Internet que hemos visto en este capítulo. Aunque ciertamente al web fue decisiva en el crecimiento y popularidad de Internet entre todo tipo de usuarios, se creó en 1990, mientras que Internet llevaba funcionando desde 1983.

4.7.2. Internet no es la nube

La «nube» es otro término común en el lenguaje cotidiano. No es raro escuchar expresiones como «tengo mis fotos en la nube» o «lo he subido a la nube» que son correctas si se refieren por ejemplo a almacenamiento remoto o a una aplicación que proporciona algún proveedor y que se controla por medio de la web.

Pero tampoco es raro que se utilice «la nube» prácticamente como sinónimo de «Internet», y eso sí es claramente incorrecto. Internet es la infraestructura de comunicaciones, y eso no incluye aplicaciones y servicios; de forma similar a como la red eléctrica no incluye los electrodomésticos, o las red de carreteras no incluye los vehículos. En todo caso hay excepciones y podemos considerar algunos servicios clave, como DNS o DHCP, parte de la infraestructura.

Cuestión aparte es la ‘computación en la nube’ (*cloud computing*), que aunque también involucra computadores remotos, implica una forma muy particular de gestionarlos y utilizarlos, que veremos más adelante. De momento bastará con que sepas que no todo lo que está en la nube es *cloud computing*.

4.7.3. Internet no es TCP/IP

Es perfectamente posible crear una nueva red o interred basada en la pila de protocolos TCP/IP, con encaminamiento dinámico o características com-

plejas al nivel de Internet, pero eso no la convierte en otra Internet. Solo hay una Internet y es la interred pública global. Es un nombre propio y por eso no puede haber más de una.

4.7.4. Internet no es un empresa u organización

Internet está compuesta por miles de redes interconectadas. Algunas de esas redes son propiedad o están bajo control de diferentes gobiernos, pero la mayoría de ellas son propiedad operadores de telecomunicaciones, proveedores de acceso (ISP) o proveedores de contenidos y servicios, entre otros. Por eso es una infraestructura descentralizada y, aunque hay actores muy importantes en su gestión, no hay una sola organización que controle todo Internet.

Sin embargo, sí que hay algunos aspectos clave de coordinación que gestiona la ICANN o los RIR como son la asignación ordenada de direcciones, nombres o los números de los Sistemas Autónomos. Sin ellos, u organismos que los sustituyeran, la funcionalidad de Internet no podría seguir creciendo ni podría adaptarse a cambios en su estructura.

Y ¿qué más?

La tecnología TCP/IP es la base de Internet. A diferencia de muchas otras, no se diseñó para reemplazar completamente a las interiores, sino para integrarse con ellas. El protocolo y el direccionamiento IP proporcionan un medio para que los mensajes que circulan por las redes LAN se puedan llevar (encapsulados) a cualquier otra red del mundo utilizando los routers IP. El protocolo IP es tan versátil y escalable que desde su creación se ha aplicado a tecnologías de enlace que ni siquiera existían cuando se diseñó. IP es una tecnología de integración de redes, que permite crear interredes heterogéneas, y eso es justamente lo que es Internet.

Aunque la interconexión es un problema importante, hay muchos otros. Los modelos de referencia OSI, TCP/IP e híbrido nos permiten desgranar esos problemas en capas y abordar sus soluciones de un modo ordenado y sistemático, y a eso dedicaremos el resto del libro.

Capítulo 5

Protocolos esenciales

Al terminar este capítulo, entenderás:

- Qué función cubre cada uno los protocolos esenciales.
- Cómo es el formato de los mensajes de esos protocolos.
- Qué es y cómo funciona la encapsulación de mensajes.
- Qué es el direccionamiento lógico y físico.

Los protocolos esenciales son las auténticas tripas de Internet, y como tal, no suelen estar a la vista de los usuarios, pero sin ellos, Internet no sería posible, o sería muy diferente. Estos protocolos han condicionado la evolución de Internet para bien y para mal. Su diseño abierto y simple es en buena medida responsable de su gran éxito, rápida adopción, flexibilidad y de la siempre cuestionada «neutralidad de la red».

Sin embargo, ese mismo diseño también es responsable de sus limitaciones y problemas, como la ausencia de mecanismos de seguridad integrados o la escalabilidad limitada en ciertos aspectos. Hay que tener en cuenta que estos protocolos fueron diseñados para una red experimental, sin siquiera plantear el gran y rápido crecimiento que tendría. Aunque la tecnología de Internet ha evolucionado mucho desde sus comienzos, y se han añadido extensiones y protocolos nuevos para cubrir deficiencias y nuevas necesidades, lo cierto es que la parte esencial sigue funcionando sobre la misma base de protocolos que se diseñaron hace más de 40 años, lo que da una idea de la calidad de su diseño.

En este libro vamos a utilizar el modelo híbrido como guía (Figura 5.1). Esta figura incluye también los protocolos más importantes de TCP/IP, que vamos a estudiar a lo largo del libro. En la capa de enlace incluye otros dos protocolos: PPP y Ethernet, que aunque no sean parte de TCP/IP son muy utilizados y parte esencial también del funcionamiento de Internet.

La implementación de estos protocolos es muy variopinta. Los protocolos de enlace se implementan típicamente en el hardware de la NIC, los de red

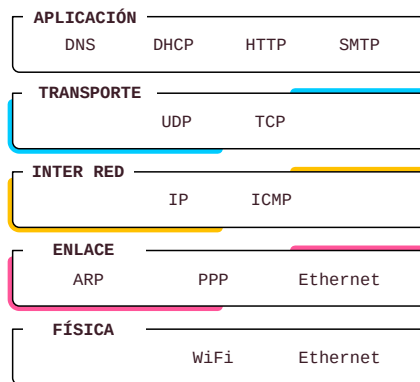


FIGURA 5.1: Modelo híbrido con los protocolos esenciales

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 el código de 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 ASIC para conseguir mayor rendimiento y menor consumo.

También aprovecharemos esta primera toma de contacto con los protocolos esenciales para explicar cómo se realiza la comprobación de la conectividad en cada capa y protocolo, ya que además de ser técnicamente relevante, ayuda a entender mejor el rol que juega cada protocolo.

La **conectividad** es la capacidad de dos o más (normalmente dos) dispositivos o procesos para intercambiar información entre ellos. Cuando existe un problema de conectividad, no es posible realizar la comunicación. La comprobación de conectividad es una actividad habitual para aislar y localizar problemas en las comunicaciones; y también como forma efectiva de monitorizar el buen funcionamiento del equipamiento y aplicaciones de red.

5.1. Encapsulación

La encapsulación es un mecanismo fundamental de la pila de protocolos. Conceptualmente consiste en colocar el mensaje completo de la capa como «carga útil» del mensaje de la capa inmediatamente inferior. Un ejemplo real sería un datagrama UDP completo que se coloca como carga útil de un

paquete IP, que a su vez se coloca como carga útil de una trama Ethernet. La Figura 5.2 muestra este ejemplo.

Como analogía se puede pensar en un sobre utilizado por el servicio postal. En el exterior del sobre se indica la información del destinatario y en el interior se coloca la carga útil. Esa carga útil podría ser perfectamente otro sobre con indicaciones específicas de la entrega y así sucesivamente. En el último sobre tendríamos el mensaje que el usuario emisor realmente quiere enviar al usuario destino.

En la práctica, los campos de un protocolo, incluido el de carga útil, no son más que una secuencia de bytes, sin embargo, la información que contienen las cabeceras de cada protocolo permite determinar en qué punto de esa secuencia se encuentra cada campo y dónde comienza cada uno de los mensajes encapsulados. Veremos ejemplos concretos de esto más adelante.

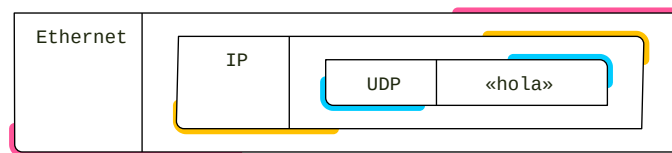


FIGURA 5.2: Ejemplo de encapsulación

Veamos cómo llevar esto a la práctica. Utilizaremos *tshark*, una aplicación de análisis de protocolos (*sniffer*), para capturar el tráfico de red. En un terminal ejecuta el siguiente comando:

```
$ sudo tshark -c 1 -V -f "udp dst port 2000"
```

Para no entrar ahora en demasiados detalles, sirva decir que lo que estamos haciendo es capturar un único datagrama UDP dirigido al puerto 2000. El comando `sudo` es necesario porque la captura de tráfico requiere de permisos especiales¹. Veremos con detalle cómo capturar y analizar tráfico en el capítulo 18.

Una vez tienes la captura en marcha, en otra consola puedes forzar la generación de un datagrama UDP con *netcat*.

```
$ echo hola | ncat -u example.net 2000
```

¹aunque `sudo` no es la forma más recomendable de hacerlo, nos sirve de momento. Consulta 18.1.1 para una explicación detallada.

Esto crea un mensaje que reproduce exactamete la Figura 5.2, es decir, coloca el texto «hola» como carga útil de un datagrama UDP, que a su vez es la carga útil de un paquete IP, que a su vez es la carga útil de una trama Ethernet. El resultado de la captura de tshark será algo parecido al Listado 5.1. Tanto en este como en posteriores listados, hemos eliminado partes de la captura que no son relevantes para el concepto que abordamos en este momento. Aún así te recomendamos hacer la captura por ti mismo en tu propio computador para ver el resultado completo.

```

1  $ sudo tshark -c 1 -f "udp dst port 2000" -nVx
2  Frame 1: 47 bytes on wire (376 bits) on interface eth0, id 0
3      Interface id: 0 (eth0)
4      Encapsulation type: Ethernet (1)
5      Frame Length: 47 bytes (376 bits)
6      [Protocols in frame: eth:ethertype:ip:udp:data]
7
8  Ethernet II, Src: f8:5f:2a:c0:ff:ee, Dst: fc:99:47:ad:f0:0d
9      Type: IPv4 (0x0800)
10
11  Internet Protocol Version 4, Src: 192.168.0.37, Dst: 93.184.215.14
12      0100 .... = Version: 4
13      .... 0101 = Header Length: 20 bytes (5)
14      Total Length: 33
15      Time to Live: 64
16      Protocol: UDP (17)
17      Header Checksum: 0x49e0 [validation disabled]
18      Source Address: 192.168.0.37
19      Destination Address: 93.184.215.14
20
21  User Datagram Protocol, Src Port: 33262, Dst Port: 2000
22      Source Port: 33262
23      Destination Port: 2000
24      Length: 13
25      UDP payload (5 bytes)
26
27  0000  68 6f 6c 61 0a                               hola.
28      Data: 686f6c610a
29
30  0000  fc 99 47 ad f0 0d f8 5f 2a c0 ff ee 08 00 45 00  ....,.]!x1..E.
31  0010  00 21 8c 61 40 00 40 11 b0 83 c0 a8 00 25 5d b8  .!.a@.@.....%].
32  0020  d7 0e cd e5 07 d0 00 0d 4d 66 68 6f 6c 61 0a  ....Mfhola.

```

LISTADO 5.1: Captura de un datagrama UDP con tshark

Se pueden apreciar 6 secciones en la captura, que corresponden respectivamente a cada uno de los protocolos involucrados y por último al mensaje completo representado en hexadecimal. Los vamos a analizar someramente ahora y después los iremos desggranando en las siguientes secciones de este mismo capítulo.

- **Línea 2:** muestra un resumen del mensaje completo. Indica la cantidad de bytes y bits capturados, la interfaz de red, el tipo de trama y

la cadena de protocolos encapsulados: `eth:ethertype:ip:udp:data`, es decir, una trama Ethernet, que contiene un paquete IP, que contiene un datagrama UDP, que contiene datos.

- **Línea 8:** aparece información de la cabecera Ethernet, con sus direcciones MAC origen y destino, y tipo de carga.
- **Línea 11:** muestra información de la cabecera IP, con las direcciones IP origen y destino y otros campos relevantes.
- **Línea 21:** ofrece información de la cabecera UDP.
- **Línea 28:** representa la carga útil del datagrama UDP, que en este caso es el texto «hola» codificado en ASCII, es decir, la secuencia hexadecimal `686f6c610a`, aunque `tshark` también lo ofrece como texto a la derecha. El punto al final de «hola.» corresponde al byte `0a` que es un salto de línea `\n`.
- Por último, aparece el mensaje completo. En el margen izquierdo muestra el offset de cada fila con un valor de 4 dígitos. En la parte central representa el contenido organizado en filas de 16 bytes, y a la derecha el mismo contenido, pero codificado como texto ASCII, siempre que corresponda con un símbolo del código; en otro caso muestra puntos.

Esto que hemos hecho enviando un texto directamente como carga útil no es lo más habitual. Normalmente el datagrama UDP contiene un mensaje de algún protocolo de aplicación (*p. ej.* DNS), pero lo hemos hecho así para que el ejemplo sea ultra simple, y de todos modos, esto es algo factible, y no contradice ninguna norma o principio, hay protocolos sencillos y reales que hacen cosas similares.

Otro detalle que merece la pena destacar es que `example.net` aunque existe, no tiene ningún servidor UDP escuchando en el puerto 2000, pero por el modo en que funciona UDP, eso no impide que podamos crear el datagrama y enviarlo. Obviamente no va a llegar a ninguna parte y esto es buena prueba de las garantías que ofrece UDP: ninguna.

5.2. Protocolos de enlace

Los protocolos de la capa de enlace de datos se encargan de mover mensajes dentro del enlace local —por ejemplo, en la LAN a la que está conectado nuestro computador. Por tanto lleva mensajes desde un computador o equipo de comunicaciones hasta sus vecinos².

²dispositivos cercanos que pueden ser alcanzados sin la intervención de intermediarios



Para el modelo TCP/IP, la capa de enlace (incluida en lo que llama «host a red») simplemente se encarga —por medio del protocolo correspondiente— de llevar paquetes IP entre vecinos de la misma red, obviando todos los detalles que ello implica. El modelo OSI por otra parte, sí que entra a estudiar todos los problemas que implica la comunicación en un enlace local.

Dependiendo del medio físico se distingue entre dos tipos de medios:

Punto a punto que permite enviar mensajes únicamente entre dos dispositivos. Un ejemplo sencillo de esto puede ser un enlace de infrarrojos.

Difusión (o *broadcasting*) se caracterizan por el uso de un medio físico compartido por más de dos dispositivos. Esto permite que un mismo mensaje pueda llegar a todos o a un subconjunto determinado de vecinos. El aire en una comunicación WiFi es 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. En función del tipo de medio físico, se pueden clasificar también los protocolos y tecnologías de enlace. Así, por ejemplo, Ethernet es un protocolo de enlace de difusión, mientras que PPP es un protocolo punto a punto.

En los medios de difusión se requiere una dirección física para determinar para qué nodo es el mensaje, mientras que en los enlaces punto a punto no es necesario, porque los mensajes solo tienen un sitio a dónde ir: el otro extremo.

Los mensajes que circulan por un enlace se llaman «tramas» (*frames*) y son contruidos por las NIC a partir de los mensajes que les llegan desde la capa superior. Normalmente, las NIC también se encargan de convertir estas tramas en señales eléctricas, ópticas o de radiofrecuencia que se transmiten por el medio físico.

Cada tecnología de enlace tiene sus limitaciones y características específicas como puede ser la velocidad de transmisión (también llamado «ancho de banda»), la distancia máxima que puede haber entre los dispositivos conectados o la cantidad máxima de dispositivos que pueden estar conectados.

5.3. Ethernet

Con mucha diferencia sobre el resto, las tecnologías LAN más utilizadas en la actualidad son Ethernet IEEE 802 y Wi-Fi que, salvando las distancias, podríamos ver como una especie de «Ethernet inalámbrica».

Ethernet cubre los detalles tecnológicos de la transmisión de señales (la capa física), pero también la funcionalidad, codificación y formato de los mensajes (la capa de enlace) que es en lo que nos vamos a centrar en este punto.

5.3.1. Interfaces de red

Para conectar nuestro computador a una LAN Ethernet o cualquier otra tecnología de enlace como Wi-Fi, Bluetooth, GSM, LTE o incluso infrarrojos, necesitamos hardware específico, es decir, una interfaz de red (NIC) que soporte Ethernet, Wi-Fi, o la tecnología correspondiente, y también un controlador (*driver*) de dispositivo que se encargue de la gestión y transferencia de datos con el SO.

Hoy en día en el ámbito del PC o *smartphone*, muchos de estos dispositivos suelen estar integrados en la placa base (especialmente Wi-Fi y Bluetooth), pero también están disponibles como periféricos externos, normalmente mediante conexión USB.



FIGURA 5.3: Interfaz de red Ethernet a USB

La forma más sencilla de ver con qué interfaces de red cuenta el nodo es el comando `ip addr`³ (del paquete `iproute2`) como en el ejemplo del Listado 5.2. Se omite la información no relevante en este momento. Puedes probar a ejecutarlo en tu propio computador para ver la salida completa en tu caso.

```

1 $ ip addr
2 1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 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 2: eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc fq_codel state UP
6     link/ether f8:5f:2a:c0:ff:ee brd ff:ff:ff:ff:ff:ff
7     inet 192.168.0.37/24 brd 192.168.0.255 scope global dynamic noprefixroute eth0

```

LISTADO 5.2: Salida del comando `ip addr`

En la salida puedes ver dos secciones diferenciadas, una especie de «fichas», para cada una de las dos interfaces de red que ha encontrado, en este caso: `lo` y `eth0`. La interfaz `eth0` corresponde a una NIC de tipo Ethernet (**línea 7**) con la dirección física: `f8:5f:2a:c0:ff:ee`.

Si hubiera una segunda NIC Ethernet se llamaría `eth1`, y del mismo modo las NIC Wi-Fi tienen nombres como `wlan0`, aunque dependiendo de la versión del SO y su configuración puede tener otros nombres como `eno1`, `enp0s32d5`, `wlp1s0`, etc.

5.3.2. Trama Ethernet

El formato de la trama Ethernet (Figura 5.4) es bastante simple.

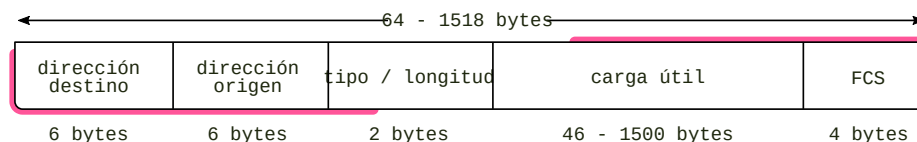


FIGURA 5.4: Formato de la trama Ethernet

Un pequeño resumen de los campos que forman la cabecera:

- Las direcciones MAC de las NIC origen y destino.
- Tipo/longitud puede indicar el tamaño en bytes del campo *payload* o bien un código que indica de qué tipo (qué protocolo) es la carga útil.

³Se puede abreviar como `ip a`.

- La carga útil (*payload*) que es de tamaño variable (mínimo 46 bytes y máximo 1500). En este campo se coloca un mensaje de la capa superior, en Internet un paquete IP, aunque puede transportar otras cosas, como ARP.
- FCS, una especie de suma de comprobación que permite al receptor verificar que la trama no ha sufrido cambios.

Observa de nuevo en la parte de la captura 5.1 que corresponde a la cabecera Ethernet.

```
Ethernet II, Src: f8:5f:2a:c0:ff:ee, Dst: fc:99:47:ad:f0:0d
Type: IPv4 (0x0800)
```

Aparecen claramente las direcciones origen y destino y el tipo, que en este caso es 0x0800 e indica que la carga útil es un paquete IP. Este campo se conoce comúnmente como *EtherType*⁴. El tamaño de la cabecera Ethernet es por tanto de 14 bytes (6 para dirección origen, 6 para dirección destino y 2 para tipo).

Obviando todos los detalles del funcionamiento de Ethernet, que no son pocos, el servicio que ofrece es simple: lleva la trama hasta la NIC con la dirección destino que aparece en la cabecera, siempre que sea un vecino de la misma LAN. Las tramas Ethernet no pueden salir nunca de la LAN en la que se crearon.

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 esa trama hasta el computador destino. La NIC emisora calcula el FCS y lo añade al final. Al llegar a su destino, la NIC receptora realiza el mismo cálculo; si corresponde con el valor FCS, la trama es correcta y será entregada a la capa superior, probablemente IP. Si el resultado de ese cálculo no coincide, la trama será descartada...y eso es todo. Ethernet no informa del error a nadie, no pide una retransmisión, no hay temporizadores ni confirmaciones. Si una trama se corrompe durante su viaje, el receptor la descartará sin más. Obviamente habrá muchas aplicaciones que no puedan tolerar ese comportamiento, pero resolver ese problema no es responsabilidad de Ethernet.

5.3.3. Direcciones MAC

En los protocolos de enlace de medio compartido (como Ethernet) se identifica cada dispositivo con una dirección física o hardware. En el caso de

⁴Puedes encontrar la lista oficial de tipos Ethernet en <https://www.iana.org/assignments/ieee-802-numbers/ieee-802-numbers.xhtml>

Ethernet, también se le llama «dirección MAC» o simplemente «dirección Ethernet».

Se trata de un número grabado por el fabricante en la memoria ROM de la NIC. La dirección es única, no existen dos tarjetas Ethernet con la misma dirección en todo el mundo, o al menos no deberían. Es lo que se llama un «identificador único global» o UUID. A pesar de eso, se utiliza únicamente para identificar esta tarjeta solo en el ámbito de la red LAN a la que está físicamente conectada. Todo esto es igualmente aplicable a una NIC WiFi y otros protocolos y tecnologías similares.

En concreto, la dirección MAC Ethernet es un número de 6 bytes que se representa como convenio como una lista de dígitos en hexadecimal separados por el carácter ':', por ejemplo: `f8:5f:2a:c0:ff:ee`. Puedes ver la dirección MAC de una interfaz de tu computador con el comando `ip`.

```
$ ip link show eth0
2: eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc fq_codel state UP
   link/ether f8:5f:2a:c0:ff:ee brd ff:ff:ff:ff:ff:ff
```

La dirección MAC está formada por dos partes:

OUI (3 bytes) Identifica al fabricante de la tarjeta. Este prefijo lo otorga la IANA. Puedes consultar la lista de todos los OUI asignados actualmente en su web⁵.

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

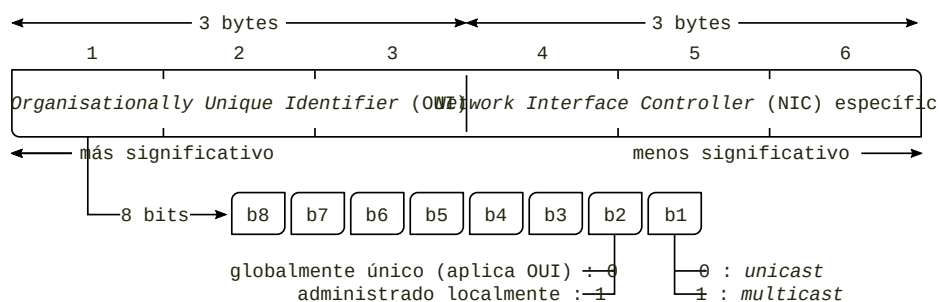


FIGURA 5.5: Formato de la dirección MAC Ethernet

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

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

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.

5.3.4. Conectividad Ethernet

La comprobación de conectividad en el caso de Ethernet consiste en verificar que la NIC está en disposición de comunicarse con un vecino de la LAN por sí mismo, sin la participación de ningún otro protocolo.

Si tu computador dispone de una NIC Ethernet normalmente estará conectada a un conmutador (*switch*), ya sea directamente o a través de una roseta en la pared. Físicamente el conector 8P8C⁶ hembra suele presentar dos LEDs (ver Figura 5.6), que ofrecen información interesante sobre el estado del enlace.

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 su color 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.

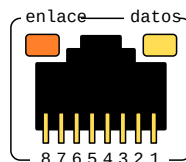


FIGURA 5.6: Conector 8P8C (RJ45) hembra de un equipo Ethernet

Aparte de la comprobación visual anterior, es posible preguntar al controlador del dispositivo con un programa específico. Uno de esos programas es `ethtool`. Lo siguiente es la salida de dicho programa para una interfaz Ethernet de un PC conectado a un conmutador.

```
1 # ethtool eth0
2 Settings for eth0:
3 Supported ports: [ TP ]
4 Supported link modes: 10baseT/Half 10baseT/Full
```

⁶El conector 8P8C se conoce más común, aunque erróneamente, como RJ45

```

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 ver, se trata de un controlador Gigabit (**línea 6**) conectado a un conmutador (**línea 14**)⁷ de 100 Mbps (**línea 14**) y el enlace de datos está correctamente establecido (**línea 23**).

5.4. PPP

PPP (Point to Point Protocol) [8] es un protocolo de enlace, pero a diferencia de Ethernet, se utiliza en enlaces punto a punto, como por ejemplo, una conexión directa entre dos routers, o entre un router doméstico y el ISP (la llamada «última milla»). También se utiliza de forma habitual sobre enlaces virtuales, túneles y VPNs.

Es un protocolo muy versátil, lo que explica sus usos tan variados:

- Tiene la capacidad de encapsular distintos protocolos de red.
- Puede configurar los detalles de esos protocolos de red (*p. ej.* la dirección IP del suscriptor) mediante los llamados NCP.
- Permite establecer, configurar, probar y cerrar el enlace, con el protocolo LCP.
- Ofrece mecanismos para autenticar al suscriptor mediante distintos protocolos de autenticación como PAP o CHAP.

El formato básico de la trama PPP es relativamente sencillo (Figura 5.7), aunque puede variar en función del enlace, opciones o protocolos que encapsule.

Por compatibilidad con HDLC los campos *Dirección* y *Control* tienen un valor fijo de 0xFF y 0x03 respectivamente. El campo *Proto* es similar al

⁷Sólo se puede establecer un enlace *full duplex* si Ethernet conmutada, es decir, el PC está conectado a un conmutador (*switch*).

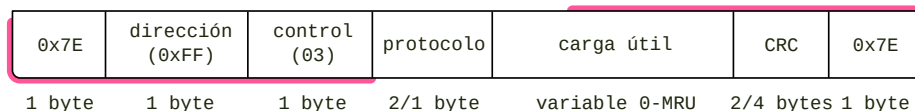


FIGURA 5.7: Formato de la trama PPP

campo *Type* de Ethernet e indica el tipo de carga útil: IP, IPX, AppleTalk, etc. El campo *Carga útil* es el mensaje encapsulado y *CRC* es un código de verificación. El flag 0x7E indica el inicio y fin de cada mensaje.

5.5. IP

IP⁸ es el protocolo de interred. Su tarea es llevar paquetes paquete IP desde un punto del planeta a cualquier otro mediante comunicación ‘extremo a extremo’ (*end-to-end*). Sin embargo, no ofrece garantías de entrega: IP es un protocolo *best-effort*. Eso significa que hará lo posible por llevar el paquete a su destino, pero bajo determinadas condiciones, lo «mejor posible» puede ser absolutamente nada.

IP es el acrónimo de ‘Internet Protocol’, pero cuidado, no significa ‘el protocolo de la Internet’, sino ‘protocolo de interred’, es decir, el que hace posible la interconexión de redes y la creación de interredes.

5.5.1. Interfaces de red

Las interfaces de red tienen una dirección lógica (una dirección IP), aparte de la dirección hardware que ya hemos visto. La dirección IP, en el caso de un dispositivo doméstico (PC, smartphone, etc.), se obtiene habitualmente de forma automática solicitándola a un servidor DHCP. Desde el punto de vista de su ámbito, se distingue entre 2 tipos de direcciones:

- **Privadas:** Se asignan a dispositivos inaccesibles desde Internet. Se usan en redes privadas y solo sirven para comunicarse con computadores dentro de la misma red o interred privada⁹.
- **Públicas:** Se asignan a dispositivos accesibles desde cualquier parte. Se asignan a computadores directamente conectados a Internet, como servidores web, correo, etc.

⁸Cuando veas escrito «IP » estamos hablando de IPv4 (IP versión 4), que sigue siendo la más utilizada hoy en día. Para referirnos a ‘IP versión 6’ siempre lo escribiremos como IPv6.

⁹Existe un mecanismo llamado NAT que permite superar parcialmente esa limitación.

Como veremos más adelante, se han definido unos rangos de direcciones específicos para cada tipo de dirección.

En el caso de una red doméstica, el proveedor de servicios (el ISP) puede asignar una única dirección IP pública al *router* doméstico, y el *router* a su vez, asigna direcciones privadas a los dispositivos de la red doméstica.

En el caso del listado 5.2, la dirección IP de la interfaz `eth0` es `192.168.0.37`, que es privada.

5.5.1.1. Interfaz *loopback*

Cualquier SO que implemente IP ofrece una interfaz de red llamada *loopback*¹⁰. Es una interfaz de red bastante especial. Se dice que es una interfaz «virtual» porque no está ligada a ninguna NIC. 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 nodo, tal como indica la **línea 4**: `scope host` del Listado 5.2.

Como no hay una NIC asociada, la interfaz *loopback* no tiene dirección física y tiene siempre la dirección IP especial `127.0.0.1`, independiente del tipo de computador o sistema operativo. El nombre simbólico para esa dirección IP es «localhost».

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

- Es de tipo bucle: `LOOPBACK`.
- Tiene una MTU de 16 436 bytes.

5.5.2. Paquete IP

El formato del paquete IP [9] se muestra en la Figura 5.8.

Veamos brevemente el significado y utilidad de cada campo:

versión

Versión del protocolo. Siempre contiene el valor 4 porque estamos describiendo el protocolo IP versión 4, que es la que se utiliza desde el nacimiento de Internet y sigue siendo la de mayor uso en la actualidad, aunque convive con la versión 6.

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

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

¹⁰que podríamos traducir como «bucle local»

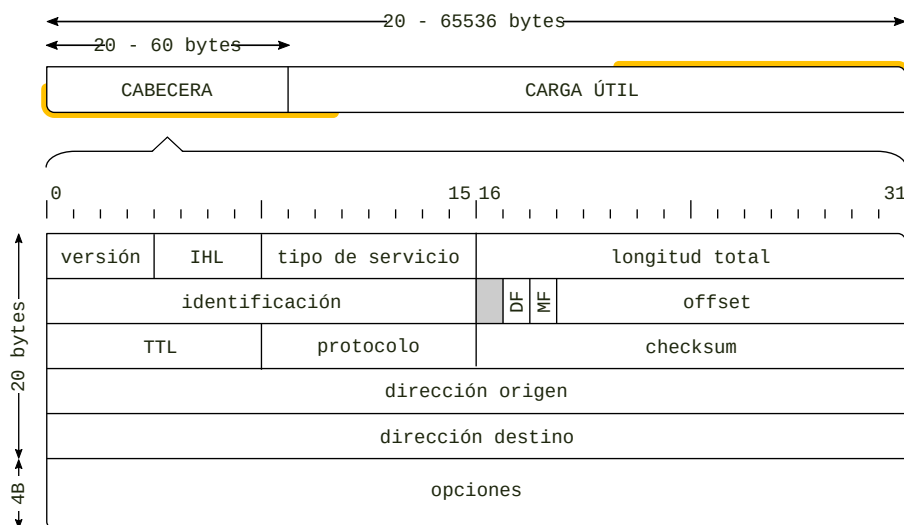


FIGURA 5.8: Formato del paquete IP

longitud total

La longitud total del paquete incluyendo cabecera y carga útil. Como es un entero de 16 bits implica que el tamaño máximo de un paquete IP es 65 635 bytes.

identificación

Un número que identifica a todos los fragmentos que proceden de 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.

offset

Indica qué posición ocupa este fragmento en el paquete original.

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

protocolo

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

checksum

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

dirección origen

La dirección IP del host que crea el paquete.

dirección destino

La dirección IP del destinatario del paquete.

Analicemos la parte de la captura 5.1 que corresponde con la cabecera IP (esta vez incluyendo más detalles).

```
Internet Protocol Version 4, Src: 192.168.0.37, Dst: 93.184.215.14
0100 .... = Version: 4
.... 0101 = Header Length: 20 bytes (5)
Identification: 0xcf0a (53002)
010. .... = Flags: 0x2, Don't fragment
...0 0000 0000 0000 = Fragment Offset: 0
Total Length: 33
Time to Live: 64
Protocol: UDP (17)
Header Checksum: 0x49e0 [validation disabled]
Source Address: 192.168.0.37
Destination Address: 93.184.215.14
```

Muestra la versión (4), la longitud de la cabecera (5=20 bytes), el identificador (0xcf0a), el flag DF (no fragmentar), el offset del fragmento (0), la longitud total del paquete (33 bytes), el TTL (64), el protocolo de la carga útil (17=UDP), la suma de comprobación de la cabecera (0x49e0), la dirección origen (192.168.0.37) y, por último, la dirección destino (93.184.215.14).

5.5.3. Direcciones IP

A diferencia de la dirección MAC, la dirección IP no viene grabada en el dispositivo sino que es asignada cada vez que el computador se conecta a una red¹¹. La dirección IP es un número de 32 bits y tiene una estructura jerárquica que la relaciona con esa red y se utiliza para identificar al computador en toda la interred, y no solo en ese enlace.

Como acabamos de ver, se representan como 4 cifras en base decimal separadas por puntos. Puedes averiguar la dirección IP de una interfaz de tu computador también con el comando `ip`.

```
$ ip addr show eth0
2: eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc fq_codel state UP
    link/ether f8:5f:2a:c0:ff:ee brd ff:ff:ff:ff:ff:ff
    inet 192.168.0.37/24 brd 192.168.0.255 scope global dynamic noprefixroute eth0
```

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

broadcast Es posible direccionar, al menos en teoría, todos los hosts de una red.

¹¹Por esto se las denomina «direcciones lógicas»

multicast Se puede direccionar un grupo arbitrario de hosts utilizando direcciones de grupo (las que empiezan por 240). Este mecanismo no es trivial ya que implica la suscripción activa de los hosts a grupos y requiere soporte en los *routers* que deben propagar la información de membresía. Para ello se utiliza un protocolo específico llamado IGMP [10] y protocolos de encaminamiento específicos.

En el capítulo 7 veremos todos los detalles del direccionamiento IP.

5.5.4. Conectividad IP

Una vez has verificado que el computador puede establecer un enlace de datos con sus vecinos puedes comprobar si es posible enviar (y recibir) tráfico IP hacia o desde otro PC, independientemente si es un vecino o está al otro lado del planeta. Las herramientas habituales para comprobar la conectividad IP se basan en el protocolo ICMP, así que las vamos a ver más adelante, en §5.7.1, cuando hayamos aprendido un poco sobre ese protocolo.

5.6. ARP

ARP (Address Resolution Protocol) es un protocolo auxiliar de IP necesario cuando un nodo conectado a un enlace de difusión (asumamos Ethernet) necesita comunicarse con sus vecinos. En este caso, el nodo emisor debe construir una trama, en la que debe colocar la dirección MAC del vecino destino. Sin embargo, el nodo emisor no conoce ese dato, pero lo que sí conoce es la dirección IP del destino.

ARP resuelve este problema. El nodo emisor envía una petición ARP a todos los vecinos (es un mensaje *broadcast*) preguntando si hay alguno que tenga asignada la IP destino. Si lo hay, el vecino que la tenga, construirá una respuesta ARP con su dirección MAC y la enviará específicamente al nodo solicitante (este mensaje es *unicast*). El vecino conoce la dirección MAC del solicitante porque aparece en la petición ARP.

El formato del mensaje ARP se muestra en la Figura 5.9. Es un mensaje de tamaño variable porque se puede utilizar con distintos protocolos de enlace y/o red, que pueden tener direcciones (físicas y lógicas) de tamaños distintos.

los campos de un mensaje ARP se listan a continuación. Para cada campo, se indica entre paréntesis el valor que tendría para el caso en que el protocolo de enlace sea Ethernet o WiFi y el de red sea IP, que es el caso más habitual.

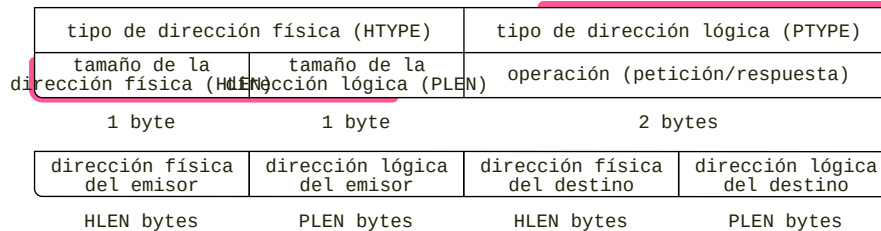


FIGURA 5.9: Formato del mensaje ARP

hardware type (HTYPE)

Es un código que indica el tipo de dirección física (Ethernet=1).

protocol type (PTYPE)

Tipo de dirección lógica (IP =0x0800).

hardware length (HLEN)

Número de bytes de la dirección física (en bytes) (Ethernet=6).

protocol length (PLEN)

Número de bytes de la dirección lógica (en bytes) (IP =4).

operation

Operación a realizar: 1=petición, 2=respuesta.

sender hardware address

Dirección física del emisor (6 bytes para Ethernet).

sender protocol address

Dirección lógica del emisor (4 bytes para IP).

target hardware address

Dirección física del destinatario.

target protocol address

Dirección lógica del destinatario.

Veamos la captura de una petición y una respuesta ARP. Lo podemos conseguir con el siguiente comando y probablemente solo hay que esperar unos segundos para que aparezcan algunas tramas. Como en las capturas anteriores, se muestran solo las partes relevantes en este momento:

```
$ sudo tshark -f arp -nVx
Frame 1: 60 bytes on wire (480 bits) on interface eth0, id 0
[Protocols in frame: eth:ethertype:arp]
Ethernet II, Src: fc:99:47:ad:f0:0d, Dst: ff:ff:ff:ff:ff:ff
  Type: ARP (0x0806)
  Padding: 000000000000000000000000000000000000000000000000
Address Resolution Protocol (request)
  Hardware type: Ethernet (1)
  Protocol type: IPv4 (0x0800)
  Hardware size: 6
  Protocol size: 4
  Opcode: request (1)
  Sender MAC address: fc:99:47:ad:f0:0d
```

```

Sender IP address: 192.168.8.193
Target MAC address: 00:00:00:00:00:00
Target IP address: 192.168.8.235

0000  ff ff ff ff ff ff fc 99 47 ad f0 0d 08 06 00 01  ...vd.....7....
0010  08 00 06 04 00 01 fc 99 47 ad f0 0d c0 a8 08 c1  .....7....
0020  00 00 00 00 00 00 c0 a8 08 eb 00 00 00 00 00 00  .....
0030  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  .....

Frame 2: 42 bytes on wire (336 bits) on interface eth0, id 0
[Protocols in frame: eth:ethertype:arp]
Ethernet II, Src: f8:5f:2a:c0:ff:ee, Dst: fc:99:47:ad:f0:0d
Type: ARP (0x0806)
Address Resolution Protocol (reply)
Hardware type: Ethernet (1)
Protocol type: IPv4 (0x0800)
Hardware size: 6
Protocol size: 4
Opcode: reply (2)
Sender MAC address: f8:5f:2a:c0:ff:ee
Sender IP address: 192.168.8.235
Target MAC address: fc:99:47:ad:f0:0d
Target IP address: 192.168.8.193

0000  fc 99 47 ad f0 0d f8 5f 2a c0 ff ee 08 06 00 01  ....7...vd.....
0010  08 00 06 04 00 02 f8 5f 2a c0 ff ee c0 a8 08 eb  .....vd.....
0020  fc 99 47 ad f0 0d c0 a8 08 c1  ....7....

```

Lo que vemos aquí es una petición ARP (frame 1) enviada por el router local (192.168.0.1) y la respuesta ARP correspondiente (frame 2), que la envía el mismo computador que está haciendo la captura (192.168.0.37). Claramente el formato de ambos mensajes es el mismo, solo cambia el valor del campo *Opcode* que indica precisamente si es una petición o una respuesta. Observa que la petición va sobre una trama broadcast (dirección destino ff:ff:ff:ff:ff:ff), mientras que la respuesta va en una trama unicast (dirigida a la dirección MAC del emisor de la petición). Quizá lo más relevante es que el campo *Target MAC address* de la petición tiene un valor cero, porque ese es precisamente el dato que desconoce y pretende averiguar. Observa también que mientras la petición incluye un relleno (*padding*) para lograr que la trama Ethernet llegue al mínimo de 64 bytes, en la respuesta no aparece. Esto es porque esta respuesta ha sido capturada antes de salir del computador. Cuando la trama se envíe, la NIC añadirá el relleno.

ARP es lo que se llama un protocolo de descubrimiento de vecinos (*neighbor discovery*) o de resolución de direcciones, ya que en la práctica permite averiguar la dirección MAC de un nodo a partir de su dirección IP. De hecho, los programas de captura de tráfico suelen resumir un mensaje de petición ARP como «Who has 192.168.0.37? Tell 192.168.0.1» y la respuesta como «192.168.8.37 is f8:5f:2a:c0:ff:ee».

Cabe señalar que los nodos no preguntan por las MAC de los vecinos cada vez que tienen que enviar una trama. El nodo dispone de una caché donde se guardan las últimas respuestas, y así puede reutilizarlas evitando tráfico innecesario.

Puedes ver la caché ARP de tu computador con el comando `arp`.

```
$ sudo arp -n
Address      HWtype  HWaddress      Flags Mask    Iface
192.168.0.200 ether    44:09:17:e0:b8:5c  C          eth0
192.168.0.1  ether    fc:99:47:ad:f0:0d  C          eth0
192.168.0.198 ether    7c:83:cb:a4:34:b7  C          eth0
192.168.0.184 ether    00:21:66:58:5c:c8  C          eth0
```

Ahora que conoces ARP puedes utilizarlo como forma de comprobar la conectividad Ethernet. Para eso puedes generar una consulta ARP con el programa `arping`. Si el vecino responde, significa que está conectado y operativo. Eso es lo que hace el siguiente comando:

```
~$ sudo arping -c2 192.168.0.1
ARPING 192.168.0.1
60 bytes from 94:83:c4:02:ec:09 (192.168.0.1): index=0 time=2.279 msec
60 bytes from 94:83:c4:02:ec:09 (192.168.0.1): index=1 time=10.207 msec

--- 192.168.0.1 statistics ---
2 packets transmitted, 2 packets received, 0% unanswered (0 extra)
rtt min/avg/max/std-dev = 2.279/6.243/10.207/3.964 ms
```

Por supuesto, esto también ocurre cuando se envía un ping ICMP (lo hace el SO), pero con `arping` ni IP ni ICMP entran en juego, de modo que se puede aislar cualquier problema adicional no relacionado con Ethernet.

5.7. ICMP

ICMP [11] también es un protocolo auxiliar de IP, que sirve para:

- Enviar notificaciones sobre problemas o errores en la entrega de paquetes.
- Realizar diagnósticos e informar sobre el estado de la red.

Los mensajes ICMP se encapsulan sobre datagramas IP, pero a pesar de ello lo consideramos un protocolo de la capa de interred¹².

El formato del mensaje ICMP es sencillo (Figura 5.10). Sin embargo, dependiendo del tipo de mensaje, puede incluir campos con significados específicos y distinta carga útil.

¹²Para muchos autores (y también para nosotros) la capa en la que se ubica un protocolo tiene que ver más con su función que con el protocolo sobre el que se encapsula.

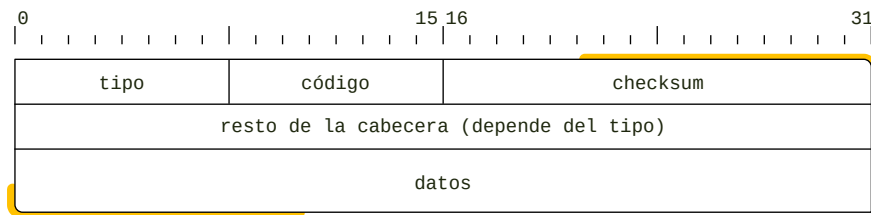


FIGURA 5.10: Formato del mensaje ICMP

Los mensajes ICMP de error se envían siempre al nodo que creó el paquete que provocó el problema. Por ejemplo, si un *router* no sabe cómo llevar un paquete a su destino final, creará un mensaje ICMP de tipo 3 (destino inalcanzable) y lo enviará al nodo que lo envió, es decir, colocará el mensaje ICMP en un paquete IP cuya dirección destino es la dirección origen del paquete problemático, y colocará su propia dirección IP (la del *router*) como dirección origen. Eso significa que el nodo sabe quién le está informando del problema. En este caso además, el mensaje ICMP incluye una copia de la cabecera del paquete IP problemático más 8 bytes de su carga útil. Así el SO puede determinar cuál de sus procesos fue el responsable del paquete problemático y, a su vez, ese proceso podría informar al usuario mediante una excepción u otro mecanismo de reporte de error disponible.

Para ilustrar el formato de la cabecera ICMP veamos la captura de una petición *Echo Request* —más conocido como *ping*— y su correspondiente respuesta. Este tipo de mensaje se puede enviar fácilmente con el programa del mismo nombre (*ping*). Aunque sencilla, resulta ser una herramienta muy útil y común para comprobar la conectividad IP, como veremos enseguida.

Por un lado, vamos a poner a *tshark* a capturar mensajes ICMP:

```
$ sudo tshark -f icmp -nV
```

Y por otro, vamos a enviar un único mensaje *ping* al nodo *example.net*:

```
$ ping -c 1 example.net
PING example.net (93.184.215.14) 56(84) bytes of data.
64 bytes from 93.184.215.14: icmp_seq=1 ttl=52 time=159 ms

--- example.net ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 159.448/159.448/159.448/0.000 ms
```

En la captura de *tshark* vemos algo parecido a esto:

```
Frame 1: 98 bytes on wire (784 bits), on interface wlp1s0, id 0
[Protocols in frame: eth:ethertype:ip:icmp:data]
Ethernet II, Src: 0c:96:e6:76:64:e1, Dst: 0a:4a:d7:39:89:ba
  Type: IPv4 (0x0800)
Internet Protocol Version 4, Src: 192.168.10.67, Dst: 93.184.215.14
  Total Length: 84
  Protocol: ICMP (1)
Internet Control Message Protocol
  Type: 8 (Echo (ping) request)
  Code: 0
  Checksum: 0x8d86 [correct]
  Identifier (BE): 65286 (0xff06)
  Identifier (LE): 1791 (0x06ff)
  Sequence Number (BE): 1 (0x0001)
  Sequence Number (LE): 256 (0x0100)
  Timestamp from icmp data: May  6, 2024 21:54:10.393995000 CEST
  Data (40 bytes)
    101112131415161718191a1b1c1d1e1f202122232425262728292a2b2c2d2e2f3031323334353637

Frame 2: 98 bytes on wire (784 bits), on interface wlp1s0, id 0
[Protocols in frame: eth:ethertype:ip:icmp:data]
Ethernet II, Src: 0a:4a:d7:39:89:ba, Dst: 0c:96:e6:76:64:e1
  Type: IPv4 (0x0800)
Internet Protocol Version 4, Src: 93.184.215.14, Dst: 192.168.10.67
  Total Length: 84
  Protocol: ICMP (1)
Internet Control Message Protocol
  Type: 0 (Echo (ping) reply)
  Code: 0
  Checksum: 0x9586 [correct]
  Identifier (BE): 65286 (0xff06)
  Identifier (LE): 1791 (0x06ff)
  Sequence Number (BE): 1 (0x0001)
  Sequence Number (LE): 256 (0x0100)
  Timestamp from icmp data: May  6, 2024 21:54:10.393995000 CEST
    101112131415161718191a1b1c1d1e1f202122232425262728292a2b2c2d2e2f3031323334353637
```

Puedes comprobar fácilmente que los valores de los campos de la cabecera ICMP corresponden con lo que hemos comentado. Puedes ver los tipos de mensaje ICMP en detalle en la sección 8.6.

5.7.1. Conectividad IP (ahora sí)

Como acabamos de ver, ping permite verificar que el nodo consultado está activo, accesible, y es capaz de recibir y enviar paquetes IP hacia y desde el nodo desde el que hemos lanzado el comando, es decir, verifica conectividad simétrica. En el caso del ejemplo anterior, está verificando que existe conectividad entre el nodo local y nodo `example.net`, pero eso no significa necesariamente que tenga la misma conectividad con otros nodos de la misma red.

Cabe destacar que ping no sirve para demostrar lo contrario, es decir, un nodo perfectamente funcional y con plena conectividad IP podría no respon-

der a *ping* por distintos motivos. Es posible configurar un cortafuegos para ignorar mensajes ICMP o específicamente mensajes *Echo Request*. Tampoco se puede usar *ping* para demostrar conectividad asimétrica, es decir, que el tráfico IP es posible en un sentido, pero no en el contrario.

En cualquier caso, se puede utilizar ICMP para comprobar la conectividad IP de otras formas y con otras herramientas. A continuación vamos a ver algunas de estas herramientas, incluyendo *ping* con más detalle, y algunas de las funciones que pueden ofrecer.

5.7.1.1. ping

Aparte de la verificación de conectividad IP, que obviamente es la funcionalidad principal, una consecuencia interesante del funcionamiento del programa *ping* es que puede calcular el RTT (Round-Trip Time) o «tiempo de vuelo», es decir, el tiempo transcurrido desde que se envió la petición hasta que se recibe la respuesta.

Prueba a ejecutar *ping* hacia tu *router* local y estudiemos con más detalle la información que ofrece.

```
1 $ ping 192.168.0.1
2 PING 192.168.0.1 (192.168.0.1) 56(84) bytes of data.
3 64 bytes from 192.168.0.1: icmp_req=1 ttl=255 time=2.02 ms
4 64 bytes from 192.168.0.1: icmp_req=2 ttl=255 time=0.730 ms
5 64 bytes from 192.168.0.1: icmp_req=3 ttl=255 time=1.66 ms
6 ^C
7 --- 192.168.0.1 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
```

Ejecutado de esta forma (sin argumentos), el programa enviará mensajes *ping* de forma indefinida (uno por segundo) hasta que pulses *control-c* (lo que ha pasado en la **línea 6**).

Las líneas 3 a 5 aparecen al recibirse cada 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). Como el nodo destino (el *router* local en este caso) es un vecino, ese tiempo 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 requerido. En la última línea aparecen los RTT mínimo, medio, máximo y la desviación estándar, que son datos interesantes para tener una idea de la latencia entre estos dos nodos y también de cómo varía con el tiempo. Por cierto, esta variación se conoce como *jitter* y es importante para cierto

tipo de aplicaciones como la telefonía IP, el streaming multimedia o los videojuegos en línea.

El programa `ping` es una de las formas más fáciles y habituales para establecer un proceso de monitorización automática para una serie de nodos que formen parte de la infraestructura de red de una organización, y de hecho prácticamente todos los sistemas de monitorización lo proporcionan de un modo u otro.

Como todos los comandos, `ping` también tiene opciones. Aquí puedes ver algunas que resultan interesantes para modificar su comportamiento:

- c Envía la cantidad de peticiones indicada 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.

5.7.1.2. `ping` «extendido»

Algunas implementaciones de `ping` proporcionan opciones para características avanzadas que, en realidad, se corresponden con campos y parámetros de los mensajes: TTL, 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.

5.8. UDP

UDP es un protocolo de transporte. Eso significa que se encarga de mover mensajes entre procesos, que lógicamente pueden estar ejecutándose en nodos diferentes. Los mensajes UDP se llaman «datagramas» y se encapsulan sobre paquetes IP. Tal como hemos visto en la sección anterior, para determinar a qué nodo se debe enviar el datagrama se utiliza la dirección IP destino; para determinar el proceso dentro de ese nodo se utiliza el puerto

destino. A esto se le llama «multiplexación» y es la que permite que distintos procesos, incluso de la misma pareja de nodos, puedan mantener flujos de tráfico simultáneos e independientes.

El puerto es un simple entero de 16 bits sin estructura. Su función es permitir la comunicación entre procesos: el proceso servidor solicita al SO vincularse a un número específico para permanecer en estado de *escucha* pasiva, es decir, ese proceso queda inactivo en espera de una petición proveniente de un cliente. Cada puerto solo puede estar vinculado a un proceso, aunque un proceso puede vincularse a múltiples puertos.

Para organizar los distintos servicios los puertos están organizados en tres rangos:

Bien conocidos Desde el 1 hasta el 1 023 están reservados para servicios comunes y muy habituales. Se requieren permisos específicos para arrancar un servidor vinculado a un puerto de este rango.

Registrados Desde el 1 024 hasta el 49 151 están reservados para servicios registrados.

Dinámicos y efímeros Desde el 49 152 hasta el 65 535 están reservados para servicios dinámicos y puertos efímeros, que son los que el SO asigna de forma automática cuando el programador no especifica uno concreto.

UDP es un protocolo muy simple. De hecho, prácticamente la única funcionalidad que ofrece sobre IP es la citada multiplexación. No tiene corrección de errores, ni control de flujo, congestión, o duplicados... Simplemente, se coloca una secuencia de bytes como carga útil del datagrama y se envía al destino como una unidad individual e independiente, y sin ninguna garantía de que vaya a llegar. Tampoco proporciona conexión o desconexión, y por eso se dice que es un «protocolo sin conexión» (*connectionless*).

A pesar de que a primera vista parezca un protocolo de poca utilidad o interés, el hecho es que UDP es un protocolo ligero y eficiente, con mínima sobrecarga. Eso lo hace adecuado para situaciones en las que no es crítico que todos los mensajes lleguen a su destino, o cuando los recursos son limitados o la sobrecarga de un protocolo más complejo es inaceptable; para transmisiones en tiempo real, transmisión de audio o vídeo en tiempo real, actualización de estado en videojuegos, monitorización o telemetría.

5.8.1. Datagrama UDP

La Figura 5.11 muestra el formato del datagrama UDP. Por supuesto la cabecera UDP también es muy simple:

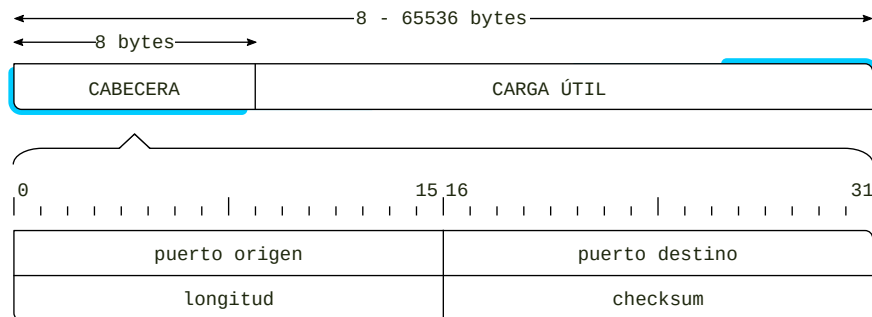


FIGURA 5.11: Formato del datagrama UDP

Puerto origen

Puerto asociado al proceso que envía el datagrama. Como no tiene por qué haber respuesta, este campo es opcional (en cuyo caso contendrá un 0).

Puerto destino

Puerto asociado al proceso al que va dirigido el datagrama.

Longitud

Longitud total del datagrama UDP incluyendo cabecera y carga útil.

Checksum

Suma de comprobación, que sirve para verificar si el datagrama ha sufrido corrupción en los datos durante la transmisión. Esto ofrece una detección básica de errores, aunque también es opcional.

Veamos una cabecera UDP que corresponde a la captura 5.1:

```
User Datagram Protocol, Src Port: 33262, Dst Port: 2000
Source Port: 33262
Destination Port: 2000
Length: 13
Checksum: 0x995d [unverified]
UDP payload (5 bytes)

0000  68 6f 6c 61 0a                hola.
Data: 686f6c610a
```

Y puedes apreciar claramente los cinco campos de los que hemos hablado. En este caso, el datagrama UDP indica que el puerto destino es 2000, el origen el 33262, tiene una longitud de 13 bytes (incluyendo la carga útil), y la suma de comprobación es 0x995d. La carga útil es la palabra «hola» seguida de un salto de línea, que codificado en ASCII y hexadecimal son los 5 bytes 686f6c610a, y más los 8 bytes de la cabecera, hacen el total de 13 bytes.

5.8.2. Conectividad UDP

La conectividad UDP implica la posibilidad de que un cliente sea capaz de enviar un datagrama UDP y el proceso servidor al que va dirigido lo pueda recibir adecuadamente.

El mensaje que acabamos de analizar corresponde con los comandos de la sección 5.1, pero no sirve para verificar conectividad UDP, ya que solo es un mensaje enviado por un cliente y capturado desde el mismo computador, ni siquiera hay servidor.

Para comprobar la conectividad vamos a usar `ncat` creando un servidor UDP en un nodo y también con `ncat` un cliente UDP en el otro nodo. El servidor se crea con:

```
alice@node1$ ncat -u -l 2000
```

Y en otra consola ejecutamos el cliente con:

```
bob@node2$ ncat -u node1 2000
```

Ahora escribe algo de texto en la consola del cliente, pulsa **ENTER** y debería aparecer en la del servidor. Si quieres comprobar la conectividad en el sentido opuesto escribe algo de texto en la del servidor, pulsa **ENTER** y debería aparecer en la del cliente.

Es importante destacar que, como en el caso de IP, es perfectamente posible tener conectividad en uno de los sentidos, pero no en el otro. Un problema de encaminamiento o un cortafuegos en uno de los dos nodos o en cualquier de los routers del camino podría descartar los datagramas UDP solo en uno de los sentidos.

5.9. TCP

TCP es «el otro» protocolo de transporte de TCP/IP. Podríamos decir que TCP es lo contrario a UDP en casi todo. Para empezar, TCP está orientado a conexión, lo que implica que antes de enviar o recibir datos, cliente y servidor obligatoriamente tienen que intercambiar información de sincronización —durante el proceso de conexión— y, de forma similar, también se aplica un procedimiento de desconexión que asegura que ambos extremos saben que el otro da por terminada la comunicación.

La conexión asegura que solo cuando ambas partes tienen la información necesaria sobre el otro puede comenzar la transmisión de datos. A diferencia de UDP, TCP garantiza la integridad de los datos, es decir, que se no se producen cambios, no hay partes ausentes ni duplicadas. Para lograr esto,

TCP utiliza un mecanismo de confirmación, temporizadores y retransmisión de los segmentos. «Segmento» es el nombre que se da a los mensajes que se intercambian en TCP. Además, incluye mecanismos de control de flujo para evitar saturar al receptor, y de congestión para evitar colapsar la red.

La comunicación TCP se comporta como un flujo continuo y bidireccional de bytes. Ambos procesos pueden enviar datos adicionales en cualquier momento y de forma indefinida. Sin embargo, los procesos no controlan cuándo se envían realmente los datos a través de la red. Es el SO el que se encarga de agrupar esos bytes en segmentos y enviarlos en el momento más adecuado para maximizar la eficiencia al mismo tiempo se aplican los mecanismos de control de flujo y congestión.

5.9.1. Segmento TCP

TCP es un protocolo bastante complejo, de modo que comprender el funcionamiento de estos mecanismos, incluso de una forma somera, requiere abordarlos cada uno por separado, y eso lo haremos en sucesivos capítulos. Por supuesto esta complejidad se ve reflejada en el formato de su cabecera, que se muestra en la Figura 5.12.

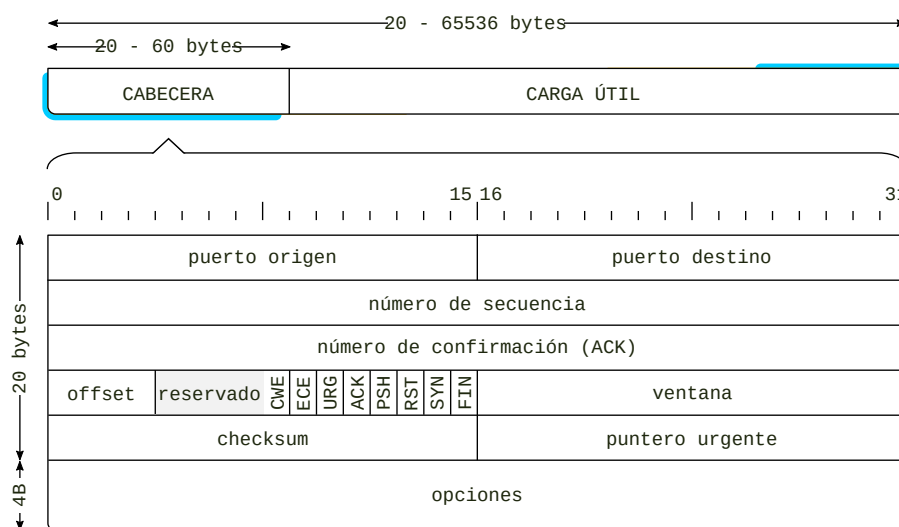


FIGURA 5.12: Formato del segmento TCP

Aunque veremos la función que desempeña cada uno de estos campos a lo largo del libro, a continuación se describen brevemente a modo de referencia. En este contexto, cuando hablemos de «emisor» nos estaremos refiriendo

al nodo que ha enviado específicamente **este** segmento, y cuando hablemos de «destinatario» para referirnos al nodo al que va dirigido.

puerto origen

Puerto asociado al proceso emisor.

puerto destino

Puerto asociado al proceso destinatario.

número de secuencia

Número de secuencia del primer byte de la carga útil del segmento.

número de confirmación

Número de secuencia del siguiente byte que el emisor espera recibir. Dicho de otro modo, es un acuse de recibo de todos los datos hasta el inmediatamente anterior al indicado.

offset

Tamaño de la cabecera expresado en palabras de 32 bits. Indica el lugar dónde comienza la carga útil, por lo que también se llama *data offset*.

flags

Los flags son un conjunto de bits que ofrecen información del segmento o sobre su emisor. Los más importantes son:

Flag	Descripción
CWR	(Congestion Window Reduced) El emisor ha recibido un segmento con el bit ECE activado y ha reducido su tasa de emisión.
ECE	(ECN Echo) El emisor sufre congestión.
URG	El campo <i>puntero urgente</i> es significativo.
ACK	El campo <i>acuse de recibo</i> es significativo.
PSH	El destino debe pasar los datos al proceso tan pronto como pueda.
RST	El emisor rechaza (o cierra) la conexión.
SYN	El emisor quiere establecer una conexión.
FIN	El emisor ha terminado de enviar datos.

CUADRO 5.1: Descripción de los flags TCP

ventana

Tamaño de la ventana del receptor. El receptor indica al emisor de cuánto espacio libre (en bytes) dispone para aceptar datos nuevos.

checksum

Suma de comprobación.

puntero urgente

Indica la posición del último byte urgente.

opciones

Cabeceras adicionales para propósitos específicos, que tienen su propio formato.

5.9.2. Capturando una conexión TCP

Como en las secciones anteriores, podemos capturar mensajes TCP con `tshark`. Vamos a replicar con TCP una situación equivalente a lo que hemos visto con UDP. En una consola ejecuta:

```
~$ sudo tshark -i lo -f "tcp port 2000"
```

En otra consola ejecuta un servidor TCP:

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

Y por último, en una tercera consola, ejecuta un cliente TCP, escribe algo y pulsa `Control-D` (que es el carácter de fin de archivo) que cerrará la conexión.

```
~$ ncat 127.0.0.1 2000
hola
C-d
```

Analicemos brevemente lo que aparece en la captura de `tshark`:

```
1 127.0.0.1 → 127.0.0.1 TCP 74 47178 → 2000 [SYN] Seq=0 Win=65495 Len=0 MSS=65495
2 127.0.0.1 → 127.0.0.1 TCP 74 2000 → 47178 [SYN, ACK] Seq=0 Ack=1 Win=65483 Len=0
↔ [...]
3 127.0.0.1 → 127.0.0.1 TCP 66 47178 → 2000 [ACK] Seq=1 Ack=1 Win=65536 Len=0
4 127.0.0.1 → 127.0.0.1 TCP 71 47178 → 2000 [PSH, ACK] Seq=1 Ack=1 Win=65536 Len=5
5 127.0.0.1 → 127.0.0.1 TCP 66 2000 → 47178 [ACK] Seq=1 Ack=6 Win=65536 Len=0
6 127.0.0.1 → 127.0.0.1 TCP 66 47178 → 2000 [FIN, ACK] Seq=6 Ack=1 Win=65536 Len=0
7 127.0.0.1 → 127.0.0.1 TCP 66 2000 → 47178 [FIN, ACK] Seq=1 Ack=7 Win=65536 Len=0
8 127.0.0.1 → 127.0.0.1 TCP 66 47178 → 2000 [ACK] Seq=7 Ack=2 Win=65536 Len=0
```

Esta vez hemos hecho una captura más sencilla. Al no añadir el argumento `-v` a `tshark`, solo muestra un resumen de algunos campos importantes de las cabeceras.

En cada línea se muestran varios datos:

- Las direcciones IP de origen y destino. En este caso es `127.0.0.1` para ambos ya que los estás ejecutando en dos consolas de tu computador y además el cliente ha conectado mediante la interfaz *loopback*.
- Los puertos origen (47178, asignado automáticamente para el cliente) y 2000 (que es dónde has puesto el servidor).
- El protocolo del mensaje más específico. En este ejemplo, el payload del segmento TCP no encapsula un mensaje de un formato que `tshark` pueda reconocer, de modo que indica TCP.
- El tamaño del segmento completo. 74 bytes para el primer segmento.
- Los flags activos. Por ejemplo, en el primero segmento está activo el flag SYN.

- El número de secuencia (Seq) y de confirmación (Ack).
- El tamaño de la ventana de recepción (Win).
- La longitud de la carga útil (Len). Solo el mensaje 4 porta datos (la cadena «hola\n») que corresponde a esos 5 bytes que indica.

Lo primero que llama la atención es que escribir un solo mensaje (con el texto «hola») ha provocado 8 segmentos. Esto es porque, como se ha dicho, TCP realiza un proceso de conexión previo y un proceso de desconexión al final.

- Los mensajes 1 a 3 corresponden al proceso de conexión.
- Los mensajes 4 y 5 corresponden al envío de la carga útil y su correspondiente segmento de acuse de recibo.
- Los mensajes 6 y 7 corresponden a la desconexión.

Para poder entender plenamente toda esta información y también otros campos de las cabeceras que ni se están mostrando aquí, tendrás que esperar a próximos capítulos donde veremos el funcionamiento detallado de TCP. Pero de momento, con lo que hemos visto aquí, ya tienes una idea de la considerable diferencia de complejidad y funcionalidad que ofrece respecto a UDP, incluso para un ejemplo tan simple como ese.

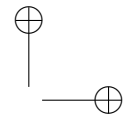
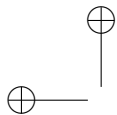
5.9.3. Conectividad TCP

El ejemplo que has realizado con `ncat` en la sección anterior para la captura es también una comprobación de conectividad TCP perfectamente válida, siempre que, lógicamente, ejecutes cliente y servidor en nodos diferentes.

Fíjate que comprobar la conectividad TCP implica obligatoriamente comprobar la simetría, ya que es imposible enviar datos si primero no se establece la conexión, y la conexión requiere segmentos en ambos sentidos. Tener conectividad en un sentido no implica necesariamente que exista en el otro. Un cortafuegos podría filtrar específicamente el segmento de conexión (SYN) de A hacia B impidiendo la conexión, pero no a la inversa. Por eso, si necesitas verificar la posibilidad de establecer la conexión en ambos sentidos, tendrás que repetir la prueba cambiando los roles como cliente y servidor.

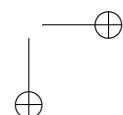
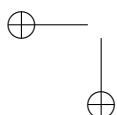
Y ¿qué más?

Esto ha sido un vistazo rápido al formato y utilidad de los protocolos principales. Es probable que como lector tengas ahora más preguntas y dudas que antes empezar el capítulo. No te preocupes, es normal. La intención era abrir camino y mostrar la parte más tangible de los protocolos: sus mensajes. Por eso hemos hecho énfasis en las capturas de tráfico real y por también por eso es muy importante que las realices por tu cuenta.



84 PROTOCOLOS ESENCIALES

En próximos capítulos veremos con más detalle el funcionamiento de cada protocolo y cómo cada campo de estas cabeceras tiene consecuencias clave en el comportamiento de la red. Nos quedan algunos protocolos importantes como DNS, DHCP, HTTP ... pero esos todavía tendrán que esperar hasta que tengas una bases sólidas sobre direccionamiento, encaminamiento y programación de redes.



Capítulo 6

Sockets

Al terminar este capítulo, entenderás:

- Qué son los sockets UDP y TCP, y cuáles son sus diferencias.
- Cómo se vincula un socket a un puerto.
- Cómo enviar y recibir datos con sockets UDP y TCP.
- Qué es la E/S parcial y cómo afecta a la programación de redes.
- Cómo delimitar los mensajes en una comunicación TCP.
- Qué es netcat y cómo usarlo para probar servidores UDP y TCP.

Un socket es un punto de conexión a la red de comunicaciones, de forma análoga a como un enchufe —que por cierto es la traducción de *socket*— es un punto de conexión a la red eléctrica. Ahí acaba el parecido porque obviamente los sockets de los que hablamos aquí no son dispositivos físicos, sino una mera abstracción de programación.

Para ser más precisos, técnicamente ‘socket’ es un API, ‘un conjunto de funciones’, para hacer posible el intercambio de datos entre dos procesos. Los sockets se parecen un poco a las tuberías (*pipes*) de POSIX; el proceso emisor escribe en un extremo y el proceso receptor lee del otro extremo.



A menudo se les llama «BSD sockets» porque el sistema operativo BSD 4.2 fue el primero en incluirlos allá por 1983. A partir de ahí se convirtió en el estándar de facto y fue adoptado por otros sistemas operativos como GNU/Linux, Windows, macOS, etc. Y así sigue siendo en la actualidad.

Ambos, tuberías y sockets, son mecanismos de IPC (Inter-Process Communication), pero a diferencia de las tuberías, que solo permiten comunicar procesos que se ejecutan en un mismo nodo, los sockets comunican procesos que se ejecutan en nodos diferentes (incluso en lugares opuestos del planeta), y también son bidireccionales, entre otras importantes diferencias.

Aunque los sockets son una abstracción relativamente simple, son la base de todo lo demás. No es descabellado decir que prácticamente todo el software que utiliza la red para comunicarse, con cualquier SO, escrito en cualquier lenguaje, y sobre cualquier plataforma, lo hace a través de sockets o de librerías que se basan en sockets.

6.1. Programación de redes

La *programación de redes* como traducción directa del inglés *network programming* engloba las técnicas, herramientas, librerías, patrones, etc. que permiten crear aplicaciones que se comunican a través de la red, o más específicamente, a la parte del desarrollo de una aplicación que se encarga de todo eso. La programación de redes engloba dos aspectos clave: los sockets y los modelos de interacción. El más utilizado de estos modelos es cliente-servidor, pero existen muchos otros como publicador-suscriptor o comunicación de pares (*peer-to-peer*).

En este capítulo nos centraremos en los sockets y las ideas básicas del modelo cliente-servidor. Vamos a tratar *solo lo esencial* acerca de los sockets porque el objetivo es que el lector adquiriera las nociones que le permitan aplicar una perspectiva pragmática a los conceptos que veremos en el resto del libro.

Dejaremos para más adelante otros asuntos más avanzados como la serialización de datos, el rendimiento, la seguridad, etc. Aún así «lo esencial» no es poca cosa. La programación de redes, y los sockets en particular, son un campo amplio, lleno de sutilezas y rincones oscuros.

6.2. La clase socket

La librería estándar de Python ofrece soporte para sockets en el módulo `socket`. En concreto este módulo proporciona una clase que también se llama `socket`. Incluye los métodos necesarios para manejar conexiones, enviar y recibir datos, etc. El constructor de la clase es:

```
socket(family, type, proto)
```

El parámetro `family` define un conjunto particular de sockets para una tecnología o pila de protocolos de red. Hay familias para Bluetooth (`AF_BLUETOOTH`), ATM (`AF_ATMPVC`), CAN (`AF_CAN`), comunicaciones infrarrojas (`AF_IRDA`) y muchas otras. En este capítulo nos limitaremos a la familia `AF_INET` que corresponde a la pila TCP/IP con IPv4.

El parámetro `type` determina el modelo de comunicación. Esencialmente hay dos tipos:

- **SOCK_STREAM**: Para comunicaciones de flujo (*stream*). La proporcionan protocolos orientados a conexión y habitualmente confiables.
- **SOCK_DGRAM**: Para comunicaciones de datagramas. La proporcionan protocolos no orientados a conexión y normalmente no confiables.

El parámetro `proto` indica qué protocolo quieres usar. Como en la familia **AF_INET** el único protocolo de tipo **SOCK_DGRAM** es UDP y el único protocolo de tipo **SOCK_STREAM** es TCP, es decir, como solo hay un protocolo de cada, este parámetro se puede y se suele omitir.

Resumiendo, puedes crear sockets UDP con:

```
socket = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
```

Y sockets TCP con:

```
sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
```

Pero como **AF_INET** es la familia por defecto, y **SOCK_STREAM** es el tipo por defecto, se puede simplificar la creación de un socket TCP a:

```
sock = socket.socket()
```

6.3. Puertos

Ambos, UDP y TCP, son protocolos de transporte, es decir, sirven para mover datos entre procesos, que es el objetivo principal de los sockets. Para lograrlo, cada socket necesita «direccionar» al otro extremo —el proceso remoto. Esto se logra con dos datos: la dirección IP del nodo remoto y el **puerto** vinculado al proceso remoto. Los puertos son enteros de 16 bits y aparecen en las cabeceras de ambos protocolos. Lógicamente, el puerto origen identifica al proceso emisor.

La tupla (IP, puerto) que identifica a un proceso globalmente en Internet se denomina «endpoint» o también «socket»¹. De este modo, cualquier flujo de comunicación entre esos dos procesos está determinado por una pareja de tuplas: (IP origen, puerto origen) e (IP destino, puerto destino). Ambas direcciones IP aparecen en la cabecera IP, y ambos puertos aparecen en la cabecera de transporte (TCP o UDP).

¹En este libro utilizaremos «endpoint» para evitar la ambigüedad de «socket».

Los routers encaminan los paquetes hacia el nodo al que corresponde la dirección destino que aparece en la cabecera IP. Cuando el paquete llega al nodo, el SO extrae el mensaje de transporte y entrega la carga útil al proceso al que corresponda el puerto destino que aparece en esa cabecera de transporte. Este mecanismo, por el cual el SO determina a qué proceso corresponden los datos en función de los puertos, se llama *multiplexación*.

Para vincular un socket a un puerto se utiliza el método `bind()`. Esta operación es imprescindible para crear un servidor porque el servidor tiene el rol pasivo de la comunicación. El servidor espera a que un cliente le contacte y por eso su endpoint debe ser conocido de antemano por el cliente. El servidor por su parte puede averiguar el endpoint del cliente en el momento que este le contacte. El Listado 6.1 muestra cómo utilizar `bind()` para un socket UDP, aunque es igual para un socket TCP.

```
import socket

sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
sock.bind(('170.10.12.4', 2000))
```

LISTADO 6.1: Vinculando un socket a un puerto UDP

El argumento de `bind()` es una tupla (ip, puerto)², no solo el puerto. Esto deber ser así porque el nodo dispone de varias interfaces de red, cada una con su dirección IP, y puedes querer que el socket quede vinculado solo a alguna de esas interfaces. Cuando quieras que el servidor sea accesible a través de cualquier interfaz del nodo —algo bastante frecuente— puedes usar la dirección especial `INADDR_ANY (0.0.0.0)`, aunque en Python se puede usar también una cadena vacía o `None`, como en `(' ', 2000)`.

```
sock.bind('', 2000)
```

El puerto 2000 no tiene nada de especial, es solo un ejemplo. Técnicamente hablando puedes usar cualquier puerto libre entre 1 y 65 535, aunque hay ciertos convenios. Los puertos inferiores a 1 024 están reservados para servicios importantes y requieren privilegios especiales. Por ejemplo, el puerto 80 es el puerto asignado para el protocolo HTTP y el 443 para HTTPS. Entre el 1 024 y el 49 151 son puertos registrados, asociados también a servicios o protocolos específicos³, aunque cualquier usuario puede ejecutar un servidor que los utilice. Por último, los puertos por encima de 49 151 son «puertos efímeros» y son asignados automáticamente por el SO cuando no se especifica, que es lo habitual en los clientes.

²Fíjate que hay otra pareja de paréntesis dentro de los de la llamada a `bind()`.

³La lista oficial completa de todos los puertos y sus asignaciones en <https://www.iana.org/assignments/service-names-port-numbers/service-names-port-numbers.txt>

Tener un número limitado de puertos implica también una cantidad limitada de procesos (en un mismo nodo) con capacidad de comunicarse con la red. Para ser exactos, un nodo puede tener un máximo de 65 535 procesos utilizando sockets TCP y otros 65 535 utilizando sockets UDP. No es un problema, es más que suficiente para cualquier uso razonable.

Desde el momento que se invoca `bind()`, se dice que el estado del puerto es «abierto». Por contra, si no hay ningún proceso vinculado a determinado puerto, decimos que está «cerrado». Es posible comprobar el estado de los puertos con el comando `ss`⁴ como se ve en el Listado 6.2. Solo un puerto abierto es susceptible de recibir y aceptar mensajes. Es el primer paso para crear un servidor UDP. El servidor TCP es algo más complejo, lo veremos enseguida.

```
$ ss -ltn
State Recv-Q Send-Q Local Address:Port Peer Address:Port
UNCONN 0 0 0.0.0.0:2000 0.0.0.0:*
```

LISTADO 6.2: Comprobando un socket UDP abierto con `ss`

El programa `ss` utiliza los servicios del SO para averiguar esta información, de modo que puedes considerarla 100 % fidedigna. Desde fuera del nodo, también es posible averiguar el estado de los puertos, aunque en ese caso los programas especializados —llamados escáners de puertos (*port scanners*)— recurren a técnicas de sondeo que no son tan fiables. Uno de los escáners de puertos más conocidos es `nmap` y lo veremos más adelante en el libro.

Para liberar un puerto, se debe invocar el método `socket.close()`, aunque también ocurre cuando el proceso termina. Cada puerto solo puede estar vinculado a un socket, y por tanto a un proceso. Si se intenta ejecutar `bind()` con el mismo puerto en otro socket se producirá un error:

```
OSError: [Errno 98] Address already in use
```

6.4. Comunicación UDP

Como vimos ya en § 5.8, UDP es un protocolo muy simple. Su cabecera solo tiene cuatro campos: los puertos origen y destino, la longitud del mensaje y un checksum (que además es opcional). La única funcionalidad real que ofrece UDP respecto a IP es precisamente esa *multiplexación* de la que hablábamos antes.

⁴`ss` sustituye a `netstat`, aunque este último sigue estando disponible aún en muchas distribuciones de GNU/Linux.

En la sección anterior decíamos que el SO asigna un puerto al socket cliente cuando contacta con el servidor. En el caso de UDP, eso ocurre cuando envía el primer mensaje. Veámoslo con un ejemplo práctico en el Listado 6.3.

```
1 $ python
2 >>> import socket
3 >>> sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
4 >>> sock.getsockname()
5 ('0.0.0.0', 0)
6 >>> sock.sendto(b'hello', ('170.10.12.4', 2000))
7 >>> sock.getsockname()
8 ('0.0.0.0', 48845)
```

LISTADO 6.3: Cliente UDP mínimo

El método `getsockname()` devuelve el endpoint *local* actual, es decir, la IP y puerto al que está asociado el socket en ese momento. El puerto 0 que aparece en la **línea 5** significa que el socket aún no tiene puerto asociado. En la **línea 6** se utiliza el método `sendto()`, que es la manera de enviar datos con un socket UDP. El primer argumento son los datos a enviar y el segundo, el endpoint destino al que deben enviarse. Al hacer esto, el SO le asigna un puerto al socket. Puedes ver cuál ejecutando de nuevo `getsockname()` como hace el listado.

Al ser UDP un protocolo sin garantías, es posible ejecutar sin quejas este código sin que de hecho exista un servidor escuchando en el endpoint destino indicado. Por supuesto, el mensaje se perderá, pero esto es lo que ofrece UDP.

Y otra cuestión importante. Como UDP es un protocolo sin conexión, no hay un destino prefijado para los datos. Por eso, es necesario indicar el endpoint en cada llamada al método `sendto()`. Eso implica que un cliente UDP puede indicar destinos diferentes para cada mensaje que envía, y también que el servidor puede recibir mensajes de clientes distintos en cada recepción.

Un servidor UDP mínimo (Listado 6.4) tiene una estructura sencilla. Después de crear y vincular el socket a un puerto, se utiliza un bucle que recibe un mensaje, lo procesa y devuelve una respuesta al emisor.

```
import socket

sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
sock.bind(('', 2000))

while 1:
    data, client = sock.recvfrom(1024)
    print(f"Se ha recibido el mensaje '{data}' desde '{client}'")
    sock.sendto(b"Enviaste {len(data)} bytes", client)
```


LISTADO 6.4: Un servidor UDP mínimo

El método `recvfrom()` es la contraparte del método `sendto()`. Su valor de retorno es una tupla con los datos recibidos y el endpoint del emisor. A continuación el código del listado imprime lo recibido en pantalla y devuelve al cliente un mensaje que indica la longitud de los datos recibidos. Tanto `sendto()` como `recvfrom()` manejan secuencias de bytes (no texto), por eso los mensajes llevan el prefijo `b`, que es el modo que tiene Python para indicar que son bytes⁵.

El argumento de `recvfrom()` indica la cantidad máxima de memoria que el programa está dispuesto a utilizar para almacenar el mensaje recibido. Si el emisor envía un mensaje mayor, el receptor solo recibirá los primeros 1024 bytes (para el caso del ejemplo) y el resto se perderá. Esto es propio de UDP al ser un protocolo orientado a datagramas. Los mensajes son y se tratan de forma independiente, y sin garantías. El programador debe decidir cuál es el tamaño de mensaje adecuado para el propósito concreto de la aplicación que está desarrollando y asegurarse de que no se truncan.

El cliente UDP mínimo (Listado 6.5) es aún más sencillo. Solo necesita enviar un mensaje y esperar la respuesta.

```
import socket

sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
sock.sendto(b'hola', ('127.0.0.1', 2000))
reply, client = sock.recvfrom(1024)
print(f"El servidor ha respondido: {reply}")
sock.close()
```

LISTADO 6.5: Cliente UDP mínimo

Este cliente envía un mensaje con el texto `'hola'` al servidor y espera a recibir una respuesta, que imprime en pantalla. Por supuesto, podría enviar y recibir varios mensajes, y como ya se ha dicho podría usar ese mismo socket para comunicarse con otros servidores diferentes durante la misma ejecución.

La implementación de los sockets por parte del SO se está encargando de la mayor parte del trabajo que implica la construcción y reconocimiento de las cabeceras y el envío de esos datos a través de la red. Recuerda que los datagramas UDP se encapsulan en paquetes IP, que a su vez se encapsulan en alguna tecnología de enlace. Esos protocolos tienen sus propias cabeceras que contienen muchos detalles como las direcciones, el tamaño de los mensajes, checksums, etc. Gracias a los sockets, el programador no tiene

⁵«bytes» es el tipo Python para secuencias de bytes.

que preocuparse de nada de todo eso. Lo único que tiene que especificar en el cliente es el endpoint del servidor y los datos a enviar. Del resto se encarga el SO.

El diagrama de la Figura 6.1 muestra la estructura y esquema de iteración típico de un cliente y un servidor UDP básicos.

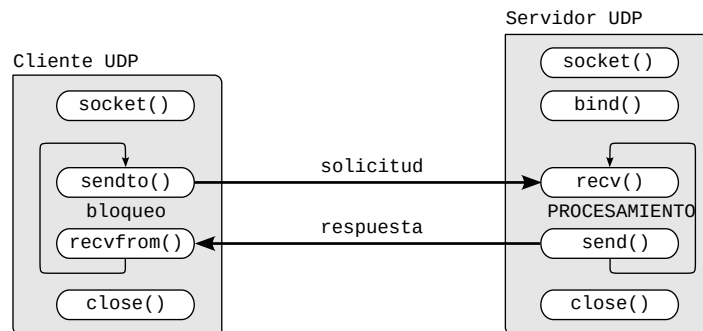


FIGURA 6.1: Diagrama de interacción de un cliente y un servidor UDP

La única diferencia significativa respecto a los ejemplos mínimos anteriores es que representa bucles para el par de invocaciones `sendto()/recvfrom()` y su contraparte en el servidor, que encaja con el comportamiento muy habitual en aplicaciones reales.

También es interesante mencionar que en el servidor, después de la recepción de un mensaje lo habitual es realizar algún tipo de procesamiento o acceso/modificación de un recurso antes de devolver un resultado o un código que indique el resultado de la operación. Mientras eso ocurre, lo habitual es que el cliente quede bloqueado en espera de la respuesta.

6.5. Servidor TCP

TCP es un protocolo mucho más complejo que UDP. El intercambio de datos no es posible si no se realiza primero un proceso de conexión exitoso. El protocolo garantiza que todos los datos llegan al otro extremo, sin errores, en el orden correcto, sin partes ausentes ni duplicadas.

Aunque se utiliza la misma clase `socket` y se usan métodos similares, o incluso los mismos que con UDP, el modo en que funcionan y la forma en que logran su objetivo es significativamente diferente. Por eso es importante conocer esas diferencias para evitar confusiones y errores.

Empecemos con un servidor TCP mínimo (Listado 6.6). Suele tener dos partes bien diferenciadas: la primera parte se llama *acceptor* y se encarga de atender nuevas conexiones —el bucle `while` con la llamada `accept()`— y la segunda parte (el manejador) se encarga de gestionar la conversación con un cliente concreto con la función `handle()`. Esta estructura se suele respetar incluso en servidores mucho más complejos.

```
1 import socket
2
3 def handle(conn):
4     data = conn.recv(1024)
5     print(f"Se ha recibido el mensaje '{data}'")
6     conn.send(b"Enviaste {len(data)} bytes")
7     conn.close()
8
9 sock = socket.socket()
10 sock.bind('', 2000)
11 sock.listen(5)
12
13 while 1:
14     conn, client = sock.accept()
15     handle(conn)
```

LISTADO 6.6: Servidor TCP mínimo

Además de la llamada a `bind()`, que funciona igual que con UDP, con TCP es necesario llamar también a `listen()`. Este método le pide al SO que ponga el socket a la escucha. Su argumento fija el tamaño del *backlog*, que es una cola en la que se insertan las conexiones en espera de ser aceptadas. Una cola de tamaño 5, como la del ejemplo, significa que podrá haber un máximo de 5 clientes a la espera. Si un sexto cliente intenta conectar, será rechazado.

Al socket `sock`, sobre el que se invoca `bind()` y `listen()`, se le llama *listening socket* o *master socket*, y solo sirve para eso: aceptar nuevas conexiones. No se puede utilizar para enviar o recibir datos.

Para aceptar conexiones (**línea 14**) se invoca el método `accept()`. El proceso queda bloqueado en ese punto hasta que un cliente inicie una conexión. Cuando eso ocurre, el método retorna una tupla con dos elementos: un nuevo socket para intercambiar mensajes con ese cliente (`conn`) y el endpoint de ese cliente (`client`).

Una vez establecida la conexión, el bucle llama a `handle()` pasando como argumento el socket `conn`. Esta función se encarga de manejar la conexión con ese cliente hasta su desconexión.

En este servidor tan simple, la función `handle()` recibe datos del cliente —con el método `recv()`—, los imprime en pantalla e invoca `send()` para

enviar una respuesta indicando la longitud⁶. Solo hay un mensaje de petición y uno de respuesta, pero como en el esquema de UDP, habitualmente la comunicación suele implicar varios mensajes. Para acabar cierra la conexión.

Una vez termina la función `handle()`, el bucle `while` llama de nuevo a `accept()` y espera al siguiente cliente. Como puedes ver, se trata de un bucle infinito, que es lo habitual en un servidor. En una versión en producción, el servidor tendría algún mecanismo para terminar de forma ordenada, habitualmente mediante un manejador de señal.

Es un programa sencillo, y todo parece correcto, salvo que... **NO LO ES**. Este código está pasando por alto la forma en que funciona realmente la comunicación TCP. Volveremos sobre el asunto enseguida, después de echar un vistazo al cliente.

6.6. Cliente TCP

La estructura del cliente TCP para enviar un único mensaje al servidor y recibir una respuesta es muy similar al cliente UDP. Se muestra en el Listado 6.7.

```
import socket

sock = socket.socket()
sock.connect(('127.0.0.1', 2000))

sock.send(b'hola')
reply = sock.recv(1024)
print(f"El servidor ha respondido: '{reply}'")
sock.close()
```

LISTADO 6.7: Cliente TCP mínimo

La diferencia más evidente es que el cliente TCP invoca `connect()` para establecer la conexión, que es la contraparte de `accept()` en el servidor. También es una llamada bloqueante, el proceso queda bloqueado hasta que el servidor acepta la petición y la conexión queda establecida. A partir de ese momento, puede comenzar el intercambio de mensajes, y como puedes comprobar a diferencia del servidor, todo se hace con el mismo socket.

Este esquema de interacción entre cliente y servidor TCP está bien representado en la figura 6.2. Como en el caso de UDP, representa una situación más general en la que cliente y el servidor intercambian varios mensajes. Por lo demás, corresponde con la idea básica de los programas cliente y servidor que acabamos de ver.

⁶como en el ejemplo de UDP

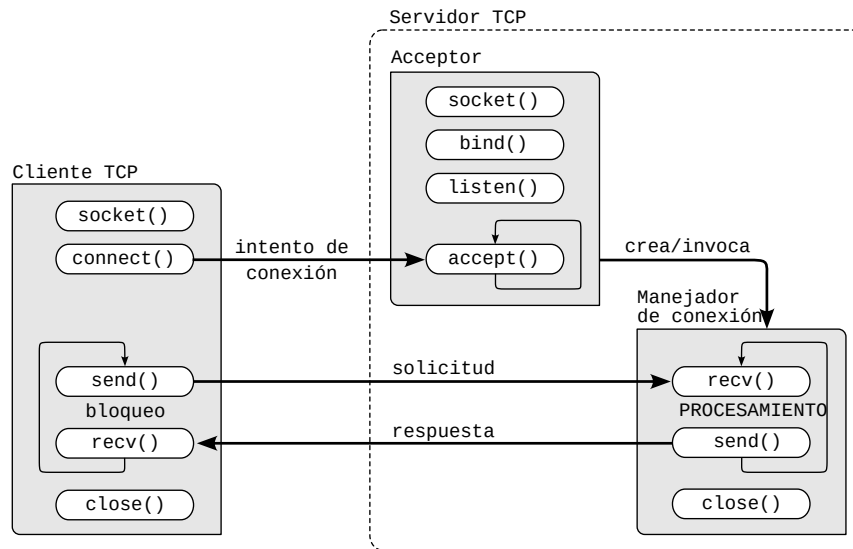


FIGURA 6.2: Diagrama de interacción de un cliente y un servidor TCP

Las dos cajas grises en el lado del servidor representan los dos elementos que hemos visto: el acceptor y el manejador de la conexión. En el Listado 6.6 el acceptor invoca directamente la función `handle`, pero como veremos más adelante en el capítulo 14, es muy habitual que esa función se ejecute en un nuevo hilo o proceso, como forma de conseguir un mayor rendimiento.

6.7. Flujos de datos y E/S parcial

En el servidor y en el cliente TCP hemos pasado por alto —a propósito— un detalle muy importante. Las primitivas `send()` y `recv()` no funcionan como en principio cabría esperar. Desde luego, distan bastante de cómo lo hacen `sendto()` y `recvfrom()` de UDP.

Por extraño que pueda sonar, invocar `send(datos)` de TCP no garantiza ni mucho menos que se vaya a enviar un segmento cuya carga útil sean exactamente esos `datos`. Aunque en condiciones de baja/media carga es probable que sí ocurra, habrá situaciones en las que varias llamadas a `send()` impliquen el envío de un único segmento (o ninguno), o en las que un único `send()` conlleve el envío de varios segmentos; a fin de cuentas, no hay correspondencia alguna entre los límites de los datos que se pasan a `send()` y la carga útil de los segmentos que se envían a la red. Aunque la probabilidad de que se den estas situaciones sea relativamente baja, el programa debe estar preparado para funcionar correctamente cuando ocurran o fallará, y

eso pasa por entender exactamente cómo funcionan estas primitivas. Y sí, eso también implica que el código que acabamos de ver en los listados 6.6 y 6.7 está **MAL**.

6.7.1. Envío TCP

La comunicación TCP trata los datos como un flujo (*stream*). Los datos que el emisor pasa como argumento a `send()` no se envían inmediatamente a la red como ocurre en UDP. En su lugar van a parar a un buffer de envío (*sending buffer*) bajo el control del SO, y cuando él lo considere oportuno, construirá un segmento e incluirá en él la parte de ese buffer que estime adecuada. Hay varias variables y condicionantes no triviales que influyen en estas decisiones y que veremos en próximos capítulos.

Además de la incertidumbre que conlleva para el programador el mecanismo de construcción y envío de segmentos, ocurre que el método `send()` funciona de un modo peculiar (o no tanto). En realidad es muy similar a la llamada al sistema `write()`^{sc} de POSIX y sufre del mismo efecto, llamado «E/S parcial» (*partial I/O*). Implica que en el momento de la invocación, el SO no asegura que pueda aceptar el bloque de datos completo que se le pasa, sino solo una parte. Puede que en ese momento el driver de dispositivo no esté listo o no tenga suficiente espacio en su buffer interno, entre otros motivos. Por esa razón, el valor de retorno de `send()`, la función `os.write()` de Python o la llamada al sistema `write()`^{sc} devuelven un entero que indica cuántos bytes pudieron escribir realmente. En realidad `os.write()` es un envoltorio Python para `write()`^{sc} así que es lógico que tenga un comportamiento análogo.

La consecuencia directa de la E/S parcial es que potencialmente son necesarias varias llamadas para asegurar que se ha enviado todo el bloque de datos. Por suerte, la solución a este problema es sencilla, al menos para envío/escritura. En un bucle se puede comprobar si quedan datos por enviar, y de ser así, invocar de nuevo `send()` con los datos que «no entraron». El bucle termina cuando el SO acepta por fin todos los datos que se pretendían enviar. Este patrón de envío es tan común que Python lo ofrece como otro método de la clase `socket` llamado `sendall()`. El Listado 6.8 podría ser una implementación simplificada a efectos explicativos.

```
def sendall(sock, data):
    sent = 0
    while sent < len(data):
        sent += sock.send(data[sent:])
```

LISTADO 6.8: implementación simplificada de `sendall()`

Una solución similar la aplican muchos lenguajes de programación cuando se escribe en archivos convencionales almacenados en disco⁷. Así, el método `file.write()`⁸ de las clases para manipulación de archivos de muchos lenguajes internamente efectúan varias invocaciones a `os.write()` hasta conseguir escribir todos los datos.

6.7.2. Recepción TCP

La recepción TCP funciona de forma análoga. Cuando se invoca `recv()`, los límites de los datos que se obtienen no tienen por qué coincidir con los segmentos recibidos y menos aún con lo que el emisor colocó en cada llamada a `send()`. También pueden ser necesarias varias llamadas a `recv()` para obtener todos los datos esperados, del mismo modo que pueden ser necesarias varias invocaciones a `os.read()` para leer la cantidad deseada de bytes desde un archivo. Por eso, el método `recv()`, la función `os.read()` y `read()`^{sc} devuelven la cantidad de bytes que realmente se han recibido/leído. Como antes, no pierdas de vista que los métodos tipo `file.read()` de las clases de manejo de archivos internamente pueden realizar varias llamadas a `os.read()` para conseguir leer la cantidad de datos que se le solicita.

Lamentablemente no hay una solución genérica para implementar un hipotético método `recvall()`, como contraparte de `sendall()`, para resolver esta situación.

El argumento de `recv()` no es de mucha ayuda en esto. Poner un número muy grande no garantiza en absoluto que vaya a obtener el mensaje completo, porque simplemente los datos esperados pueden no haber llegado aún. Como en el caso de UDP, ese argumento simplemente le dice al SO cuanto espacio como máximo estás dispuesto a utilizar para almacenar esos datos. A diferencia de UDP, si los datos recibidos no caben en ese espacio, no se pierden, sino que se quedan en el buffer de recepción (*receiving buffer*) a la espera de que invoques `recv()` de nuevo.

Lo importante aquí es que es responsabilidad del programador delimitar los mensajes y decidir hasta cuando debe seguir llamando a `recv()` y eso no siempre es sencillo. Depende completamente del objetivo y finalidad de la aplicación, o siendo más precisos, del protocolo de aplicación. Si la aplicación envía mensajes de una longitud fija, es sencillo, solo hay que recibir hasta conseguir esa cantidad de bytes. Pero si los mensajes son de longitud variable o impredecible, la cosa se complica.

⁷Utilizamos la expresión ‘en disco’ como una forma común para referirnos al almacenamiento secundario, a pesar de que la tecnología actual no involucra discos.

⁸Usamos *file* para referirnos en genérico a clases para manipulación de archivos. Python llama a esto *file object*, aunque no existe ningún tipo `file`.

En nuestro pequeño servicio contador de caracteres (Listado 6.6), el objetivo del servidor es informar de la longitud del mensaje que recibe, y obviamente es desconocido para él. Eso significa que el servidor no puede saber cuántas veces debe llamar a `recv()` para obtener el mensaje completo. Al cliente le pasa algo parecido cuando recibe: no puede saber qué tamaño tendrá el mensaje de respuesta.

El problema de fondo es que el programador diseña su aplicación para intercambiar mensajes que tienen sentido para el objetivo del programa. Algo como:

```
cliente: "hola"
servidor: "Enviaste 4 bytes"
```

Pero TCP no entiende nada de eso. Para TCP todo es una única secuencia de bytes que abarca toda la conexión y que tiene que llevar de un lado al otro del modo más eficiente posible. Para TCP no hay mensajes individuales, solo un flujo continuo de datos en cada sentido.

Para reconciliar ambos enfoques aparentemente enfrentados, el programador debe incluir algo en el mensaje que le permita, en el otro extremo, saber cuándo ha recibido un mensaje completo (desde el punto de vista de la aplicación). Y es crítico, porque si no recibe suficiente, el mensaje estará incompleto; y si intenta recibir demasiado, el proceso quedará bloqueado porque no hay más datos en ese momento.

framing

El proceso que determina el comienzo y fin de cada mensaje dentro de un flujo continuo de datos en función de la lógica propia de cada aplicación específica se denomina *framing* en inglés, y se puede traducir como *delimitación de mensajes*.

6.7.3. Bandera de fin de mensaje

Para el propósito del contador de caracteres, una posible solución sería incluir un *token* especial al final del mensaje a modo de bandera. Por ejemplo, tanto cliente como servidor podrían enviar un carácter de nueva línea (`\n`) al final de cada mensaje y el receptor simplemente tiene que seguir recibiendo hasta que aparezca. El listado 6.9 muestra una versión corregida del cliente, que aplica esta solución, y utiliza `sendall()` para el envío.

```
import socket

def read_message(conn):
    data = bytes()
    while (i := data.find(b'\n')) == -1:
        chunk = conn.recv(1024)
        if not chunk:
            return data
```



```

        data += chunk
        line = data[:i]
        return line

sock = socket.socket()
sock.connect(('127.0.0.1', 2000))

sock.sendall(b'hola\n')
reply = read_message(sock)

print(f"El servidor ha respondido: {reply.decode()}")
sock.close()

```

LISTADO 6.9: Cliente TCP recibiendo un mensaje con bandera de fin
 /socket/tcp-client-flag.py

Del otro lado, el listado 6.10 muestra una versión corregida del servidor. Este tiene en cuenta lo que hemos visto sobre E/S parcial, incluye la bandera de fin de mensaje y el manejador de señal para la finalización adecuada. Con esto, si pulsas **Ctrl+C** en la consola en la que se está ejecutando el servidor, la shell enviará al proceso la señal **SIGINT** y el servidor ejecutará la función `int_handler()` que se encarga de cerrar el socket y terminar ordenadamente el proceso. Aunque añadir este tipo de manejador es ciertamente una práctica muy conveniente en código de producción, por legibilidad no los incluiremos en los ejemplos del libro.

```

import socket
import signal

def int_handler(sig, frame):
    print("\nSIGINT recibido. Cerrando socket y saliendo...")
    sock.close()
    exit(0)

def read_message(conn):
    data = bytes()
    while (i := data.find(b'\n')) == -1:
        chunk = conn.recv(1024)
        if not chunk:
            return data
        data += chunk
    line = data[:i]
    return line

def handle(conn):
    line = read_message(conn)
    print(f"Se ha recibido el mensaje: {line}")
    conn.sendall(f"Enviaste {len(line)} bytes\n".encode())
    conn.close()

sock = socket.socket()
sock.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
sock.bind('', 2000)
sock.listen(5)

```

```
signal.signal(signal.SIGINT, int_handler)

while True:
    try:
        conn, client = sock.accept()
        handle(conn)
    except OSError:
        break
```

LISTADO 6.10: Servidor TCP recibiendo un mensaje con bandera de fin

[📄/socket/tcp-server-flag.py](#)

La función `read_message()` que busca la bandera de fin de mensaje es frágil porque podría leer más de lo debido y provocar pérdida de datos, pero de momento sirve para ilustrar este ejemplo trivial. Más adelante en este mismo capítulo veremos una solución más robusta usando el método `file.readline()`.

6.7.4. Tamaño del mensaje en la cabecera

Otra solución más flexible y robusta es incluir la longitud del mensaje en su cabecera. De ese modo no se necesita elegir un carácter especial como bandera. Esta solución es de las más habituales en los protocolos de aplicación (*p. ej.* HTTP).

Hasta este momento ni siquiera había una cabecera en nuestro ejemplo del contador de caracteres, solo se enviaba la carga útil y, con la solución anterior, la bandera de fin. Como este es un protocolo⁹ muy simple, puede tener una cabecera también muy simple. El mensaje tendrá el formato `<longitud>:<datos>` siendo `<longitud>` un número en decimal que indica la cantidad de bytes que mide el resto del mensaje, *p. ej.* `17:you all everybody`. La cabecera está formada solo por `<longitud>:` y el resto es cuerpo¹⁰.

El listado 6.11 muestra un fragmento de código en el que se realiza la recepción conforme a ese formato de mensaje. Fíjate que en este caso no hay problema en que el carácter `:` aparezca en el cuerpo del mensaje, solo el primero funciona como separador entre cabecera y cuerpo.

```
size = bytes()
while b':' not in reply:
    size += sock.recv(1)

size = int(size[:-1])

body = bytes()
while len(body) < size:
```

⁹Sí, técnicamente estamos diseñando un protocolo de aplicación.

¹⁰Ya dije que sería una cabecera simple.

```
body += sock.recv(size - len(body))
```

LISTADO 6.11: Recepción TCP con una mensaje con formato <longitud>:<datos>

Tanto esta solución con longitud en la cabecera como la bandera de fin en realidad son soluciones sencillas. Pero date cuenta que el verdadero problema es que aunque estas puedan ser soluciones razonables para este caso, pueden no ser adecuadas en otras situaciones. ¿Qué ocurre si el carácter que hemos elegido como bandera forma parte del mensaje? ¿Y si no es aceptable añadir nada al mensaje? ¿Y si la cabecera debe ser de tamaño fijo o es crítico mantener el tamaño del mensaje al mínimo? Este tipo de problemas son comunes en la programación de redes y es lo que justifica la necesidad de protocolos de aplicación y la razón por la que hay tantos y tan variados.

6.8. Sockets como archivos

La mayoría de los métodos de la clase `socket` que hemos estado usando son «envoltorios» para las llamadas al sistema `socket()`^{sc}, `connect()`^{sc}, `send()`^{sc} cuyos prototipos aparecen en `<sys/socket.h>` de la librería estándar del lenguaje C.

No es casualidad que hayamos estado haciendo analogías entre sockets y archivos. En los sistemas POSIX se aplica un principio que reza «todo es un archivo» ya sea un dispositivo de caracteres o de bloques (un disco), tubería, terminal, zona de memoria, entrada y salida estándar, etc. Por supuesto, los sockets deben considerar muchas situaciones que no ocurren con archivos convencionales y por eso `recv/send` disponen de un argumento opcional `flags` que permite modificar su comportamiento. Pero, en la mayoría de situaciones las primitivas `read/write` son equivalentes a `recv/send` y, de hecho, cuando se programa en C, lo son.

En Python no existen los métodos `socket.read()/socket.write()`, pero si hay dos opciones que permiten tratar a los sockets como archivos en caso de que convenga.

El método `socket.fileno()` devuelve el descriptor de archivo (un entero) asociado al socket. Con este descriptor puedes usar la primitivas `os.read()` y `os.write()` para recibir y enviar datos. Esto no cambia la forma de trabajar con el socket porque estas funciones también aplican la E/S parcial. Sin embargo, disponer del descriptor del archivo te da la posibilidad pasar el descriptor a un subproceso o de utilizar código de librería que no está pensando para sockets, pero acepta descriptores de archivo.

```
>>> sock.fileno()
12
>>> os.write(sock.fileno(), b'hello')
```

La otra opción es `socket.makefile()`. Este método crea un objeto de estilo archivo (un *file object*) para manipular el socket:

```
>>> sock.makefile('r')
<_io.TextIOWrapper name=5 mode='r' encoding='UTF-8'>
```

El método `sock.makefile()` tiene parámetros similares a los de la función `open()`.

- `mode`: 'r', 'w', 'rb', 'wb', etc.
- `buffering`: tamaño del buffer, o 0 sin buffer.
- `encoding`: Para hacer (de)codificación automática (si se ha indicado modo texto).
- `newline`: Para indicar qué carácter se usa como salto de línea (consulta `open()` en la documentación de Python).

Y como archivo de alto nivel, sus métodos tienen el comportamiento habitual, es decir, ocupándose de la E/S parcial haciendo varias llamadas a `os.read()/os.write()` si es necesario.

- `file.read(size)` lee `size` bytes.
- `file.readline()` lee hasta que reciba el carácter `newline` indicado en `makefile()`.
- `file.readlines()` lee todas las líneas hasta el final de la conexión.
- `file.write()` envía `data` completo, que es equivalente a `socket.sendall()`.
- `file.flush()` fuerza la escritura (envío) de los datos pendientes.

Pero recuerda que para que todo funcione, el protocolo de la aplicación que estas creando debe asegurar que es factible que lo que esperas recibir (sea por cantidad o formato) puede llegar eventualmente sin producir un bloqueo.

Al tener disponible el método `readline()`, puedes implementar una versión más robusta de la recepción del mensaje con bandera de fin que hemos escrito en §6.7.3.

```
import socket

sock = socket.socket()
sock.connect(('127.0.0.1', 2000))

sock.sendall(b'hola\n')
reply = sock.makefile().readline()

print(f"El servidor ha respondido: {reply}")
```

```
sock.close()
```

LISTADO 6.12: Cliente TCP recibiendo un mensaje con bandera de fin usando `file.readline()`

Aparte de ahorrarnos el escribir la función `read_message()`, se encarga de convertir a texto (`reply` es una cadena ya decodificada) y si quedan datos después del salto de línea lo guarda para la siguiente invocación, algo que no hacía nuestra función.

6.9. Gestión explícita de buffers

Como corresponde a un lenguaje dinámico como Python, la memoria necesaria para los buffers de las llamadas a `send()`, `recv()` y otras se reserva y libera automáticamente. Veamos como ejemplo un emisor que envía a un receptor un archivo en bloques de 1024 bytes sobre un flujo TCP. Date cuenta que en este ejemplo no importa quién es el servidor y quién el cliente, la transferencia se puede hacer en ambos sentidos exactamente de la misma forma. En los siguientes fragmentos no se incluye la creación de los sockets, conexión o desconexión, solo la transferencia.

Emisor

```
with open('outgoing.data', 'rb') as f:
    while 1:
        chunk = f.read(1024)
        if not chunk:
            break
        sock.sendall(chunk)
```

Receptor

```
with open('incoming.data', 'wb') as f:
    while 1:
        chunk = sock.recv(1024)
        if not chunk:
            break
        f.write(chunk)
```

FIGURA 6.3: Transferencia de un archivo creando buffers nuevos en cada iteración

En cada iteración de estos bucles, `read()` y `recv()` crean un buffer interno de 1024 bytes para realizar la correspondiente llamada al sistema. Después crean el buffer `chunk` con los datos que han conseguido leer/recibir. Repito: ¡en cada iteración! Esto puede suponer una ineficiencia nada despreciable, especialmente en plataformas con recursos limitados.

Tanto para `read()` como para `recv()`, el 1024 indica el tamaño del buffer que crean¹¹ y determina la cantidad máxima de bytes que pueden retornar. Sin embargo, como hemos visto en §6.7, no trabajan exactamente igual. El método `file.read()` siempre devolverá 1024 bytes a menos que no queden tantos en el archivo; mientras que `recv()` puede devolver cualquier cantidad entre 1 y 1024 aunque la conexión siga activa y queden muchos

¹¹La destrucción de todos esos buffers es tarea del recolector de basura.

miles de bytes por llegar. Si en lugar de usar *file.read()/file.write()* usases *os.read()/os.write()*, los dos bloques de código tendrían una semántica análoga.

Sin embargo, es posible evitar tanta creación y destrucción de buffers. Python ofrece una alternativa que permite crear un único buffer y reutilizarlo en cada iteración del bucle mientras dure la transferencia. Quedaría algo así:

Emisor	Receptor
<pre>buffer = bytearray(1024) view = memoryview(buffer) with open('outgoing.data', 'rb') as f: while 1: nbytes = f.readinto(buffer) if nbytes == 0: break sock.sendall(view[:nbytes])</pre>	<pre>buffer = bytearray(1024) view = memoryview(buffer) with open('incoming.data', 'wb') as f: while True: nbytes = sock.recv_into(buffer) if nbytes == 0: break f.write(view[:nbytes])</pre>

FIGURA 6.4: Transferencia de un archivo reutilizando un único buffer

El tipo `bytearray` es similar a `bytes`, pero mutable. Por eso se puede reutilizar en cada iteración para almacenar los datos procedentes del archivo en el envío o desde el socket en la recepción. Como puedes ver, los métodos `readinto()` y `recv_into()`¹² son variantes diseñadas para este propósito.

El tipo `memoryview` permite acceder a una parte del buffer sin copiar nada, por lo que es mucho más eficiente que la primera alternativa en la que *file.read()* y *recv()* creaban buffers. La variable `nbytes` indica en ambos casos cuántos bytes se han podido obtener realmente. Como el propósito del código es enviar y recibir un archivo, realmente no supone ningún problema si `socket.recv_into()` no recupera la misma cantidad de datos en todas las iteraciones.

6.10. Fin de la comunicación TCP

En los listados anteriores ya has visto que tanto el cliente como el servidor invocan `close()` para cerrar el socket, y con ello la conexión si es TCP. El receptor TCP necesita conocer esta situación. Cuando invoca `recv()`, si el otro extremo ya ha cerrado la conexión, el método devolverá una secuencia vacía. De ese modo el receptor puede terminar también de forma ordenada. Y si estaba en un bucle recibiendo mensajes sucesivos, puede interrumpirlo como corresponda.

¹²También existe un `recvfrom_into()` para sockets de datagramas.

El siguiente listado ilustra este comportamiento. Es un receptor que va concatenando los datos que va recibiendo en el buffer `reply` y sale del bucle al recibir una secuencia vacía (`if not data`).

```
reply = bytes()
while True:
    data = sock.recv(1024)
    if not data:
        break
    reply += data
```

El socket TCP dispone de una forma más fina que el método `close()` para controlar el cierre de la conexión: el método `shutdown()`. Este acepta 3 valores:

- `socket.SHUT_RD`: El proceso ya no va a invocar `recv()`. Eso le dice al SO que a partir de ese momento puede descartar cualquier mensaje entrante sin llegar a almacenarlo en el buffer. Podrá seguir enviando mensajes.
- `socket.SHUT_WR`: El proceso ya no va a invocar `send()`. Eso implica el envío de un mensaje con el flag FIN al otro extremo. Podrá seguir recibiendo mensajes.
- `socket.SHUT_RDWR`: Es la combinación de las dos anteriores.

Cuando el emisor invoca `shutdown(SHUT_WR)`, el receptor experimenta el mismo comportamiento descrito anteriormente: el método `recv()` devuelve una cadena vacía. Es decir, el receptor no puede distinguir si el emisor cerró con `SHUT_WR` o `close()`. En todo caso el proceso debería invocar `close()` para liberar el socket aunque haya invocado `shutdown()` previamente.

6.11. Manejo de errores

En los ejemplos anteriores —y en la mayoría de los que encontrarás en el libro— no se han incluido manejadores de errores y excepciones. La razón para esto es proporcionar listados más compactos y fáciles de explicar y entender. Sin embargo, en un código destinado a producción es fundamental tratar los errores de forma adecuada, porque de lo contrario el programa puede comportarse de forma errática o simplemente terminar abruptamente. Lógicamente, este tratamiento de errores es especialmente importante para los servidores, donde no puedes permitir que la operación incorrecta de un cliente provoque la caída del servidor.

La tabla 6.1 muestra los errores más habituales cuando se opera con sockets y conexiones.

Excepción	Situación
<code>socket.gaierror</code>	Error en la resolución de nombres.
<code>TimeoutError</code>	<code>connect()</code> llevó demasiado tiempo (incluso para sockets bloqueantes).
<code>BrokenPipeError</code>	Se invocó <code>send()</code> cuando el otro extremo ya había desconectado.
<code>ConnectionAbortedError</code>	La conexión se abortó durante la ejecución de <code>connect()</code> o <code>accept()</code> y no se llegó a completar.
<code>ConnectionResetError</code>	Se invocó <code>send()/recv()</code> cuando el otro extremo ya había cerrado de forma abrupta (RST en lugar de FIN).
<code>ConnectionRefusedError</code>	Se invocó <code>connect()</code> a un puerto cerrado.

CUADRO 6.1: Errores comunes en programación de sockets

Estos errores aparecen como excepciones que heredan de la clase `OSError` (Figura 6.13). Conocer la jerarquía es útil porque permite al programador decidir lo genérico o específico que quiere hacer el manejador.

```

Exception
├─ OSError (== socket.error)
│   ├── socket.gaierror
│   ├── TimeoutError
│   └─ ConnectionError
│       ├── ConnectionAbortedError
│       ├── ConnectionRefusedError
│       ├── ConnectionResetError
│       └─ BrokenPipeError

```

LISTADO 6.13: Jerarquía de excepciones de socket

Por ejemplo, si capturas `ConnectionError`, eso incluye todas las causas de error de la conexión.

```

while 1:
    try:
        conn, client = sock.accept()
    except ConnectionError as e:
        print(f"Error de conexión {e} con el cliente {client}")
        continue

```

Estas excepciones son «nombres propios» para números de error (`errno`) específicos de la excepción `OSError`, pero hay muchos otros valores de `errno` que no tienen un nombre específico. En esos casos, el programador puede capturar la excepción y comprobar el valor para determinar el problema concreto.

```

try:
    [...]
except OSError as e:
    if e.errno == errno.ECONNREFUSED:

```



```

    print("Conexión rechazada")
    elif e.errno == errno.EHOSTUNREACH:
        print("Host inalcanzable")
    elif e.errno == errno.ENETUNREACH:
        print("Red inalcanzable")
    else:
        print(f"Error desconocido: {e}")

```

A continuación puedes ver algunos de ellos. Están disponibles como constantes en el módulo `errno`.

nombre	errno	descripción
EDESTADDRREQ	89	Se requiere dirección de destino.
EPROTONOSUPPORT	93	Protocolo no soportado.
EAFNOSUPPORT	97	Familia de direcciones no soportada.
EADDRNOTAVAIL	99	Dirección no disponible.
ENETDOWN	100	La red está caída.
ENETRESET	102	La conexión se ha perdido por un fallo de red.
EISCONN	106	El socket ya está conectado.
ENOTCONN	107	El socket no está conectado.
EHOSTUNREACH	113	No se puede determinar una ruta hacia el host.
ENETUNREACH	101	La red no es alcanzable.
ETIMEDOUT	110	El tiempo de conexión se ha agotado.
ECONNREFUSED	111	El destino rechaza el intento de conexión.
ECONNRESET	104	El destino ha cerrado la conexión abruptamente.

CUADRO 6.2: Códigos `errno` relacionados con errores de red

6.11.1. Context manager

Los sockets de Python se pueden usar como *context managers*, es decir, con la clausula `with`. Esto ayuda a gestionar automáticamente los recursos asociados, ahorrando al programador la invocación de `close()` y capturando excepciones comunes. El Listado 6.14 muestra un servidor para el servicio Echo que aprovecha esa característica. Tal como indica su nombre, el servidor Echo devuelve al cliente los mensajes que este le envía.

```

import socket

def handle(conn):
    while 1:
        data = conn.recv(1024)
        if not data:
            break

        conn.sendall(data)

with socket.socket() as sock:
    sock.bind(('', 7))
    sock.listen(5)

```

```
while 1:
    conn, client = sock.accept()
    with conn:
        handle(conn)
```

LISTADO 6.14: Servidor de Echo TCP con *context manager*

Una vez el servidor acepta una conexión, recibe mensajes del cliente y los devuelve con `sendall()`. Si `recv()` retorna una cadena vacía, significa que el cliente ha desconectado. Entonces el bucle `while` de `handle()` termina con `break` y el servidor espera la siguiente conexión. Es interesante destacar que para el caso concreto de Echo utilizar `recv()` sin más no es un problema. El servidor no necesita identificar límites en los datos que recibe. Simplemente devuelve lo que recibe. Da igual si eso ocurre en bloques pequeños o grandes, iguales o diferentes, y al cliente tampoco le va a afectar, porque todo es un único flujo.

6.12. Netcat

El programa `netcat` es, a pesar de su sencillez, una de las herramientas más potentes y versátiles que existen en el campo de la programación, análisis y manipulación de redes y servicios TCP/IP. Es un recurso imprescindible para profesionales de la programación, administración o gestión de redes, expertos en seguridad y hackers.

Existen muchas implementaciones de `netcat` desarrolladas por distintos fabricantes, con sus propias características, pero todas ellas comparten una misma funcionalidad básica y simple que se resume en:

- `netcat` crea un socket TCP o UDP, que puede ser cliente o servidor en función de los argumentos que se le indiquen.
- Una vez establecida la comunicación, envía por el socket todo lo que lea de su entrada estándar y a la vez, envía a su salida estándar todo lo que reciba por ese mismo socket.

Esto lo convierte en un cliente o servidor absolutamente genérico, que combinado con la potencia de la *redirección* que ofrece la `shell` (§ 2.6) permite proporcionar conectividad de red a cualquier programa que consuma y genere datos de su entrada y salida estándar, respectivamente. Por eso se podría decir que `netcat` es un «socket» para usar desde línea de comandos.

Para que todos los usos de `netcat` que se hacen a lo largo del libro sean homogéneos, vamos a usar siempre la implementación de `insecure.org`, que se llama `ncat` y está disponible en la mayoría de distribuciones GNU/Linux en el paquete del mismo nombre.

6.12.1. Sintaxis básica

Aunque `ncat` tiene muchas opciones y funcionalidad, en la gran mayoría de las situaciones se utilizan dos comandos básicos:

- Crear un servidor TCP: `ncat -l <puerto>`
- Crear un cliente TCP: `ncat <nodo> <puerto>`

El parámetro `-l` (*listen*) indica que se trata de un servidor. Para crear un servidor o un cliente UDP simplemente añade el argumento `-u` en ambos casos.

El siguiente es el uso más básico, lo que acabamos de ver, que funciona como una especie de chat tremendamente básico, y que permite a cualquier de los dos extremos enviar y recibir en cualquier momento. Lógicamente, debes ejecutar primero el servidor y luego el cliente. Después puedes escribir en la entrada de cualquiera de los dos y el texto aparecerá en el otro extremo. En este caso, ambos se estarán ejecutando en tu PC, pero por supuesto puede hacerse en nodos diferentes.

Cliente	Servidor
<pre>\$ ncat 127.0.0.1 2000 hola[ENTER]</pre>	<pre>\$ ncat -l 2000 hola</pre>

FIGURA 6.5: Uso básico de `ncat`: servidor y cliente TCP

Veremos unos cuantos ejemplos para ilustrar las muchas posibilidades de `ncat`, pero ten en cuenta que para casi todos existen programas específicos que pueden lograr el mismo objetivo con más posibilidades y, sobre todo, más seguridad.



Prueba esos ejemplos únicamente si sabes lo que estás haciendo. Algunos pueden suponer un problema para la integridad de tu sistema y tus datos.

Transferencia de archivos

Aplicando redirección puedes enviar un archivo (tanto texto como binario) del cliente al servidor y viceversa. Suponiendo que tienes el archivo `song.mp3` en el nodo cliente, puedes ejecutar lo siguiente:

Cliente	Servidor
<pre>\$ ncat sidney 2000 < song.mp3</pre>	<pre>\$ ncat -l 2000 > copia-song.mp3</pre>

FIGURA 6.6: Transfiriendo un archivo con ncat

Servidor Echo

ncat puede ejecutar cualquier programa conectando directamente el socket que crea con la entrada/salida de ese programa. Si el programa que ejecutas es `cat` conseguirás un comportamiento equivalente al servicio Echo, es decir, devolver al cliente todo lo que reciba de él:

Cliente	Servidor
<pre>\$ ncat sidney 2000 hola[ENTER] hola</pre>	<pre>\$ ncat -l 2000 -e /bin/cat</pre>

FIGURA 6.7: Servidor Echo con ncat

Servidor *daytime*

También muy básico. Cuando el cliente conecta y sin enviar nada, el servidor que emula el servicio *daytime* le envía la fecha y hora actual y cierra la conexión. Para lograrlo, basta con ejecutar `date` en lugar de `cat`:

Cliente	Servidor
<pre>\$ ncat sidney 2000 jue 24 abr 2025 23:50:01 CEST</pre>	<pre>\$ ncat -l 2000 -e /bin/date</pre>

FIGURA 6.8: Servidor daytime con ncat

Lógicamente, puedes hacer lo mismo con cualquier programa que produzca una salida.

Shell remota

De forma similar a los ejemplos anteriores, si el programa que ejecutas con `ncat` es una shell, esta ejecutará los comandos que se le envíen y devolverá el resultado de la ejecución de esos comandos (Figura 6.9). Y para tener una *shell inversa* simplemente ejecuta la shell en el cliente (Figura 6.10).

Desde luego esta no es la forma más conveniente de ejecutar una shell remota en un entorno de producción, para eso existe `ssh`. Sin embargo es habitual encontrar cosas parecidas en software malicioso. Se suele utilizar

Ciente	Servidor
<pre>\$ ncat sidney 2000 hostname[ENTER] sidney</pre>	<pre>\$ ncat -l 2000 -e /bin/bash</pre>

FIGURA 6.9: Shell remota con ncat

Ciente	Servidor
<pre>\$ ncat sidney 2000 -e /bin/bash</pre>	<pre>\$ ncat -l 2000 hostname[ENTER] sidney</pre>

FIGURA 6.10: Shell remota inversa con ncat

como *backdoor* y normalmente explota alguna vulnerabilidad que permite al atacante puede introducir comandos en la máquina vulnerable.

Streaming multimedia

Del mismo modo que se envía el contenido de un archivo para guardarlo en disco, se puede consumir directamente por el receptor. Si dispones de un programa que reproduce audio o vídeo desde su entrada estándar (como *mplayer*) puedes enviar un archivo compatible que se irá transfiriendo a medida que se consume.

Ciente	Servidor
<pre>\$ ncat sidney 2000 mplayer -</pre>	<pre>\$ ncat -l 2000 < song.mp3</pre>

FIGURA 6.11: Streaming multimedia con ncat

Web

Puedes servir una página web con ncat, si bien un cliente estándar espera que envíes también la cabecera HTTP. Puedes usar este archivo `index.html` poco ortodoxo porque incluye la cabecera HTTP de respuesta:

```
HTTP/1.1 200 OK
Content-Type: text/html
Content-Length: 96

<!DOCTYPE html>
<html>
  <head><title>Ejemplo</title></head>
  <body>
    <h1>Hola desde ncat</h1>
  </body>
</html>
```

Lo puedes servir con el comando:

```
$ ncat -l 2000 < index.html
```

Y cargarlo con **firefox** o cualquier otro navegador web.

```
$ firefox http://127.0.0.1:8080
```

Fíjate que la consulta HTTP GET aparece en la salida del servidor.

```
GET / HTTP/1.1
Host: 127.0.0.1:8080
User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:128.0) Gecko/20100101 Firefox/128.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: es-ES,es;q=0.8,en-US;q=0.5,en;q=0.3
Accept-Encoding: gzip, deflate, br, zstd
Connection: keep-alive
Upgrade-Insecure-Requests: 1
```

Proxy

Puedes utilizar **ncat** para redirigir una conexión entrante a otro servidor, es decir, otro puerto y/u otro nodo:

```
$ ncat -l 2000 -e "ncat example.net 3000"
```

El tráfico recibido en el puerto 2000 de este nodo se redirige al nodo **example.net:3000**. Permite incluso que el flujo entrante sea UDP pero la redirección sea a un servidor TCP o viceversa!

Medir el ancho de banda

En este caso, puedes usar un cliente que envía datos a la mayor velocidad posible —leyendo del dispositivo **/dev/zero**— y el servidor que los recibe los redirige hacia el programa **pv** que calcula la velocidad de llegada. Con esto puedes medir la velocidad máxima (el ancho de banda) entre ambos nodos.

Cliente	Servidor
<pre>\$ ncat sidney 2000 < /dev/zero</pre>	<pre>\$ ncat -l 2000 pv > /dev/null 2,84GiB 0:00:04 [1,35GiB/s] [<=>]</pre>

FIGURA 6.12: Midiendo el ancho de banda con **ncat**

Imprimir un archivo PostScript

Este ejemplo funciona con impresoras que soportan el estándar AppSocket/JetDirect, que son la mayoría de las que se conectan por Ethernet o WiFi y, obviamente, debe estar habilitado en su configuración. El archivo debe estar en formato PostScript, pero lo puedes convertir fácilmente desde PDF con `pdf2ps`¹³.

```
$ pdf2ps documento.pdf
$ cat documento.ps | ncat ip.de.la.impresora 9100
```

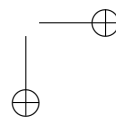
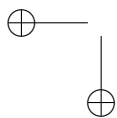
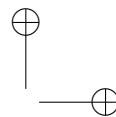
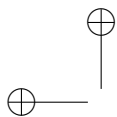
Y ¿qué más?

Hemos visto cómo los sockets proporcionan la base de la programación de redes en prácticamente cualquier sistema operativo y lenguaje de programación. Hemos visto cómo crear servidores y clientes UDP, un protocolo que proporciona comunicaciones sin conexión, rápidas, ligeras pero nada fiables. También servidores y clientes TCP, que proporcionan un flujo de datos orientado a conexión, con entrega garantizada y corrección de errores, incidiendo también en la importancia de la multiplexación y los puertos, o las implicaciones de la entrada/salida parcial y la gestión eficiente de los buffers.

Los sockets son en realidad solo el punto de partida. Hay muchos temas pendientes, que cubriremos en los siguientes capítulos, y que también son fundamentales para entender y construir aplicaciones de red: los modelos cliente-servidor y publicación-suscripción, la serialización de datos, el diseño de protocolos de aplicación, la gestión eficiente de múltiples conexiones concurrentes, etc. Además, conocer los sockets te va a ayudar a entender los detalles del funcionamiento del control de flujo, de errores, el control de congestión, etc., que veremos en los siguientes capítulos.

También la seguridad es un tema fundamental, y muchos de los ataques se aprovechan de errores relacionados con la programación de sockets o de su uso incorrecto por parte del programador.

¹³Paquete `ghostscript`



Capítulo 7

Direccionamiento IP

Al terminar este capítulo, entenderás:

- Qué es una dirección IP y cuál es su formato.
- Para qué sirve la máscara de red.
- Qué es el direccionamiento con clases y sus limitaciones.
- Qué es el direccionamiento sin clases y la técnica VLSM
- Cómo calcular el direccionamiento para una topología dada.

7.1. Qué es una dirección IP y su formato

Una dirección IP es un identificador único asignado a cada dispositivo de la interred. Se trata de un número entero de 32 bits, por lo que teóricamente el espacio de direcciones es de 2^{32} direcciones, o 4 «gibidirecciones»¹, es decir, concretamente 4.294.967.296 direcciones. Aunque parezcan muchas (y ciertamente lo son) por varias razones que veremos, no todas las direcciones están disponibles para asignar a dispositivos, hasta el punto que desde hace unos años hay una apremiante escasez de direcciones IP.

A diferencia de las direcciones físicas, las direcciones IP tienen una estructura jerárquica. Como norma general podemos considerar dos partes²: la primera es el prefijo de red (*network ID*) e identifica la red a la que pertenece el dispositivo. La segunda es el identificador de nodo (*host ID*) y sirve para eso, indica qué nodo es dentro de esa red. La Figura 7.1 muestra el formato de la dirección IP.

El prefijo de red es imprescindible para que los routers puedan hacer su trabajo. Esencialmente la función de un router es decidir por cual de sus interfaces debe enviar cada paquete que recibe. Para eso, dispone de una tabla (la tabla de encaminamiento) que recorre buscando cuál de sus filas

¹Nos tomamos la licencia de usar el prefijo «gibi» del SI que indica 2^{30} , como en las unidades de memoria.

²Más adelante veremos que puede tener más.

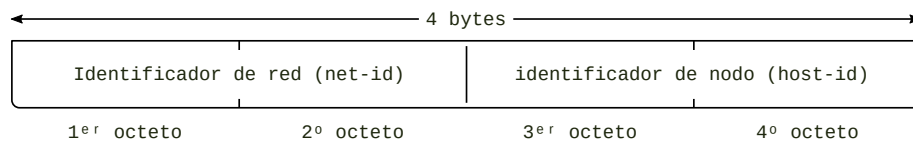


FIGURA 7.1: Formato de la dirección IP

encaja con la dirección IP destino que aparece en el paquete. El prefijo de red permite que la tabla pueda incluir las direcciones de las redes conocidas, en lugar de tener que incluir todos los posibles nodos destino. De ese modo las tablas pueden tener unas decenas de filas en lugar de millones.

7.2. Máscara de red

Para determinar qué parte de la dirección corresponde al prefijo de red y cuál al identificador de nodo se necesita la **máscara de red**. Es simplemente otro número de 32 bits que, en binario, se ve como una secuencia de unos seguida de una secuencia de ceros, aunque normalmente se representa en notación decimal, como una dirección IP. Así, una máscara formada por 16 unos seguidos de 16 ceros, en decimal sería 255.255.0.0.

Para obtener el prefijo de red se realiza una operación **AND** a nivel de bits entre la dirección y la máscara. En el ejemplo, dada la dirección 150.20.45.76, y la máscara 255.255.0.0, el resultado que se obtiene al hacer el AND es 150.20.0.0. El prefijo de red es 150.20 (la parte que ha «superado» la máscara), y el identificador de dispositivo es 45.76 (la parte eliminada por la máscara) (ver Cuadro 7.1). La dirección 150.20.0.0 (la que tiene todos ceros en la parte de identificador de nodo) representa a la red completa y por ello se conoce como **dirección de red**.

Relacionado con esto, también tenemos la **dirección broadcast**. Es la dirección que representa a todos los dispositivos de la red, y se obtiene también a partir del prefijo de red, pero con todos los bits restantes a 1, en lugar de 0. En el ejemplo, la dirección de broadcast de la red 150.20.0.0 para la máscara aplicada es 150.20.255.255.

	decimal	1 ^{er} octeto	2º octeto	3 ^{er} octeto	4º octeto
Dirección	150.20.45.76	10010110	00010100	00101101	01001100
Máscara	255.255.0.0	11111111	11111111	00000000	00000000
Dir. de red	150.20.0.0	10010110	00010100	00000000	00000000
Dir. broadcast	150.20.255.255	10010110	00010100	11111111	11111111

CUADRO 7.1: Ejemplo de uso de la máscara de red

Como en total tienen que ser 32 bits, cuando necesitamos una red grande con muchos nodos, el prefijo será corto, mientras que si es una red pequeña, el prefijo será largo. Esto también implica que si creamos demasiadas redes grandes, quedará poco espacio para las pequeñas, y viceversa.

En todo caso, una organización cualquiera no puede decidir por su cuenta qué prefijo usar para su red. Para hacer un reparto organizado y evitar conflictos, existen organizaciones específicas como la ICANN o sus delegados regionales (las RIR) que se encargan de asignar —previo pago, por supuesto— estos prefijos de red a las organizaciones que los soliciten.

Pero, ¿cómo se sabe cuál es la máscara de red que hay que aplicar? Hay dos respuestas a esta pregunta, la que se definió originalmente (que llamamos «direccionamiento con clases») y la que se utiliza en la actualidad, que llamamos «direccionamiento sin clases». Veremos ambas a continuación.

7.3. Direccionamiento con clases

Para hacer un reparto racional entre redes grandes y pequeñas, la IETF originalmente definió un sistema que llamamos direccionamiento con clases o *classfull addressing* [9], que divide el espacio de direcciones en cinco clases nombradas desde la A a la E, aunque únicamente las clases A, B y C se utilizan para direccionamiento unicast.

clase	prefijo (bits)	máscara	# redes	direcciones por red
A	0	255.0.0.0	128	16 777 216
B	10	255.255.0.0	16 384	65 536
C	110	255.255.255.0	2 097 152	256

CUADRO 7.2: Direccionamiento con clases

Con este sistema, dada una dirección IP se puede determinar directamente a qué clase pertenece, pero también mucha otra información útil. Veamos un ejemplo, con la dirección 161.67.21.13:

- Es una dirección de clase B porque los dos primeros bits de 161 en binario son 10.
- Por ser de clase B, le corresponde una máscara 255.255.0.0.
- Su dirección de red es 161.67.0.0.
- Su dirección de broadcast es 161.67.255.255.
- Su rango de direcciones asignables va de 161.67.0.1 a 161.67.255.254, es decir $2^{16} - 2$ direcciones.

Hablamos de «direcciones asignables» precisamente porque, ni la dirección de red ni la de broadcast, se pueden asignar a una interfaz de red, y por eso se deben descontar del total.

7.3.1. Subnetting

El direccionamiento con clases tiene varios problemas. Hay pocas redes de clase A y B, porque son demasiado grandes (sobre todo las clase A), y las de clase C son demasiado pequeñas, más a día de hoy. Esto provocaba un gran desperdicio de direcciones porque muchas organizaciones que solicitaron bloques clase A y B hace años solo utilizaban una mínima parte, pero nadie más las puede aprovechar.

Para tratar de resolver esto, se desarrolló el concepto de *subnetting* o más formalmente FLSM. Consiste en dividir una red cualquiera en subredes más pequeñas. Estas nuevas subredes son útiles para la misma organización que ya poseía la red original. Desde un punto de vista administrativo, permite a la organización delegar la gestión de cada subred al departamento o unidad a la que se le asigna. Eso facilita la gestión de los servicios, visibilidad hacia el exterior y privilegios de acceso específicos de cada subred, adaptándose a las necesidades, restricciones y políticas de cada departamento.

Para crear subredes, se toman bits de la parte del identificador de host y se utilizan para direccionar las subredes. A este nuevo campo se le llama «prefijo de subred» o *subnet ID* (consulta la Figura 7.2).

Identificador de red (net-id)	Identificador de subred (subnet-id)	identificador de nodo (host-id)

FIGURA 7.2: Formato de la dirección IP con *subnetting*

Si se toman n bits para el prefijo de subred, se obtendrán 2^n subredes. Siempre que se aplica *subnetting* se obtiene una potencia de dos de subredes. Eso significa que si, por ejemplo, se necesitan 3 subredes, habrá que crear al menos 4 y una de ellas quedará sin uso. Obviamente, tomar bits del identificador de host para el de subred, implica que las subredes resultantes serán más pequeñas, es decir, habrá menos direcciones para asignar a nodos.

Lo interesante del *subnetting* es que se aplica solo desde el punto de vista de la organización. Desde fuera, para el resto de Internet, el conjunto se sigue viendo como la red original completa. Nadie fuera de esa red necesita saber que se ha aplicado *subnetting* y mucho menos cómo se ha hecho, en cuantas subredes se ha dividido, etc.

Inicialmente el primer y último bloque que se obtiene al realizar la división causaba problemas en el encaminamiento porque esas direcciones se podían confundir con la dirección de red y broadcast de la red original. Para evitarlo, se prohibía el uso de estos dos bloques, llamados *subnet-zero* y *subnet-all-ones*. Es una limitación muy grave, ya que en el caso de usar pocos bits para el prefijo de subred, se podían perder muchas direcciones. En el ejemplo anterior implicaría perder la mitad del espacio de direcciones original. Más tarde la RFC 1878 [12] especificó la forma de evitar esta limitación. A día de hoy la mayoría de equipos de red y sistemas operativos lo soportan sin problema.

7.3.2. Ejemplo de *subnetting*

Veamos un ejemplo sencillo. Partimos de una red con dirección 150.20.0.0, que es de clase B (máscara 255.255.0.0), es decir, tenemos 16 bits para direccionar nodos. Vamos a dividirla en 4 subredes. Utilizaremos los 2 primeros bits del tercer octeto como prefijo de subred y los 14 restantes como identificador de nodo. La tabla 7.3 muestra las subredes resultantes. Para cada subred se muestra el tercer octeto en binario, la dirección de subred y la dirección de broadcast. Para todas ellas, la máscara de subred será 255.255.192.0.

subred	tercer octeto	dir. de subred	dir. broadcast
0	0000 0000	150.20.0.0	150.20.63.255
1	0100 0000	150.20.64.0	150.20.127.255
2	1000 0000	150.20.128.0	150.20.191.255
3	1100 0000	150.20.192.0	150.20.255.255

CUADRO 7.3: Ejemplo de *subnetting*

Existen muchas herramientas para realizar el cálculo de subredes con *subnetting*. Una de las más sencillas es *sipcalc*. En el Listado 7.1 se muestra un ejemplo para la misma división que hemos visto en el ejemplo anterior.

```
$ sipcalc -bi 150.20.0.0/16 -s /18
-[ipv4 : 150.20.0.0/16] - 0

[CIDR]
Host address^^I^^I- 150.20.0.0
Host address (decimal)^^I- 2517893120
Host address (hex)^^I- 96140000
Network address^^I^^I- 150.20.0.0
Network mask^^I^^I- 255.255.0.0
Network mask (bits)^^I- 16
Network mask (hex)^^I- FFFF0000
Broadcast address^^I- 150.20.255.255
```

```

Cisco wildcard^I^I- 0.0.255.255
Addresses in network^I- 65536
Network range^I^I- 150.20.0.0 - 150.20.255.255
Usable range^I^I- 150.20.0.1 - 150.20.255.254

[CIDR bitmaps]
Host address^I^I- 10010110.00010100.00000000.00000000
Network address^I^I- 10010110.00010100.00000000.00000000
Network mask^I^I- 11111111.11111111.00000000.00000000
Broadcast address^I- 10010110.00010100.11111111.11111111
Cisco wildcard^I^I- 00000000.00000000.11111111.11111111
Network range^I^I- 10010110.00010100.00000000.00000000 -
                    10010110.00010100.11111111.11111111
Usable range^I^I- 10010110.00010100.00000000.00000001 -
                    10010110.00010100.11111111.11111110

[Split network]
Network^I^I^I^I- 150.20.0.0      - 150.20.63.255
Network^I^I^I^I- 150.20.64.0     - 150.20.127.255
Network^I^I^I^I- 150.20.128.0    - 150.20.191.255
Network^I^I^I^I- 150.20.192.0    - 150.20.255.255

```

LISTADO 7.1: Ejemplo de uso de sipcalc

7.4. Direccionamiento sin clases

Aunque la técnica de *subnetting* facilitó un uso más eficiente, no era suficiente. Años más tarde se modificó el sistema de direccionamiento para proporcionar una mayor flexibilidad, sobre todo para el espacio que aún quedaba por asignar. Se llamó a esto «direccionamiento sin clases» o *classless addressing*, aunque la designación técnica es «encaminamiento inter-dominio sin clases» o CIDR (Classless Interdomain Routing) [13].

En el direccionamiento sin clases la máscara de red no está limitada a 8, 16 o 24 bits correspondientes a las anteriores clases A, B y C, sino que puede tener cualquier tamaño, de modo que, es posible tener prefijos de red de cualquier tamaño, por supuesto siempre dentro de los 32 bits del tamaño de la dirección IP.

Con la llegada del direccionamiento sin clases, los bits iniciales de la dirección IP ya no tienen el significado especial que tenían en A, B y C. Eso quiere decir que ya no se puede saber qué máscara aplicar si disponemos solo de la dirección. Y sin la máscara no se puede determinar la dirección de red o broadcast. Por ese motivo, es necesario conocer explícitamente la máscara junto con la dirección.

Para facilitar esta tarea, se introdujo una sintaxis más compacta: la llamada «notación CIDR». Consiste simplemente en añadir el carácter / seguido del número de bits que forman el prefijo de red. Por ejemplo, la

dirección 161.67.21.13 con máscara 255.255.0.0 se puede escribir como 161.67.21.13/16.

7.4.1. VLSM

VLSM es la generalización del concepto de *subnetting*. Permite dividir una red en subredes de diferentes tamaños —de ahí su nombre. Las máscaras varían de tamaño entre subredes. Esto permite un aprovechamiento mucho mayor del espacio de direccionamiento.

En un caso real, el objetivo de un reparto de direcciones es parte esencial del diseño de la infraestructura de comunicaciones de una organización. No se trata simplemente de dividir sin más el espacio de direcciones disponible en cierta cantidad de bloques, como hemos visto en el ejemplo de *subnetting*. Lo normal es que una organización que disponga de un determinado bloque deba proporcionar direcciones a distintos departamentos, o usos específicos, como servidores, impresoras, telefonía IP, etc.

Por eso, la organización necesita determinar primero cuáles son las necesidades de direccionamiento de la empresa y, a partir de ahí, diseñar un esquema de direccionamiento. En ese sentido, es importante tener en cuenta las expectativas de crecimiento de cada uno de los departamentos contando con el margen de direcciones libres requerido dentro de cada bloque, dado que una vez en explotación, cambiar el esquema de direccionamiento puede ser una tarea compleja.

Aunque todo esto es igualmente importante con el direccionamiento con clases y *subnetting*, VLSM resuelve mucho mejor este problema ya que ofrece mayor flexibilidad y reduce notablemente el desperdicio de direcciones.

7.4.2. Bloques /30

VLSM resulta especialmente importante cuando es necesario asignar direcciones a redes pequeñas. Y la red más pequeña posible es una necesidad habitual: direccionamiento para enlaces punto a punto, como los que se utilizan para interconectar dos routers. En ese caso, solo se necesita una dirección para cada extremo del enlace. Se utiliza en ese caso la subred de tamaño mínimo, que es la que tiene una máscara de 30 bits. Un bloque de este tipo solo tiene 2 bits para *host ID* y por tanto tiene 4 direcciones: la de red, la de broadcast y dos direcciones asignables. Por ejemplo, el bloque 170.10.20.160/30 aplicado a un enlace serie (ver Figura 7.3) tiene las siguientes direcciones:

Idealmente esta necesidad la deberían cubrir los bloques /31, pero siguiendo las reglas establecidas, las únicas dos direcciones que ofrece son la de

Dirección de red: 170.10.20.160/30
 Dirección asignable 1: 170.10.20.161/30
 Dirección asignable 2: 170.10.20.162/30
 Dirección de broadcast: 170.10.20.163/30

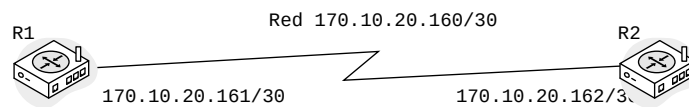


FIGURA 7.3: Direccionamiento IP de un enlace serie

red y la de broadcast, de modo que no quedaría ninguna para asignar a un nodo. Sin embargo, la [14] establece que es posible utilizar un bloque /31 precisamente para este propósito y solo para este propósito, bajo la premisa de que en un enlace punto a punto no se necesita una dirección de broadcast. Siguiendo el ejemplo anterior, el bloque 170.10.20.160/31 tendría las siguientes direcciones:

Dirección asignable 1: 170.10.20.160/31
 Dirección asignable 2: 170.10.20.161/31

En todo caso, ten presente que aunque la especificación no es nueva, puede haber equipos de red y SO soportan que no soportan esta posibilidad.

Y por último, las direcciones /32 no definen una subred, sino que identifican exclusivamente un nodo individual. Precisamente se utiliza esa máscara para enfatizar que se trata de un nodo concreto y no de una red. Por tanto, usar direcciones como 170.10.20.160/32 y 170.10.20.161/32 en los extremos de un enlace punto a punto no sería correcto, ya que sin configuración específica adicional habría problemas de encaminamiento: por ejemplo, el sistema operativo no podría resolver direcciones mediante ARP ni determinar que hay un vecino conectado directamente.

7.4.3. Ejemplo de VLSM

Partiendo de la dirección 70.50.0.0/18, vamos a definir el esquema de direccionamiento para cubrir las necesidades de la topología de la Figura 7.4, incluyendo los enlaces serie que interconectan los routers.

Asumiendo que estas cantidades de nodos expresan las necesidades de cada red considerando el posible crecimiento futuro, el proceso de división debería minimizar la cantidad de direcciones que sobran (no se asignen a nodos) en cada bloque. Al mismo tiempo, es muy conveniente que los blo-

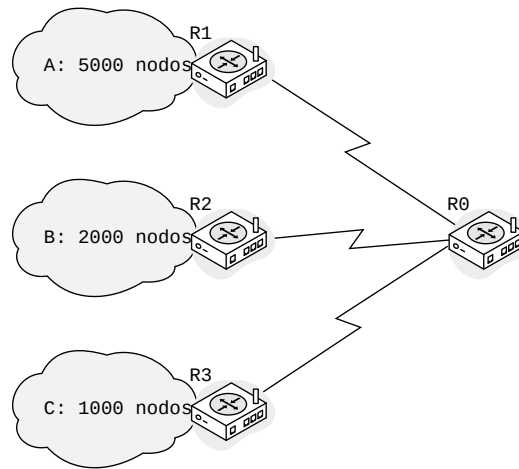


FIGURA 7.4: Topología de ejemplo para VLSM

ques sin asignar sean lo mayores posible, para que conseguir así un mejor aprovechamiento en el futuro.

Lo primero que haremos es determinar las necesidades de la topología. Veamos por ejemplo la red A que requiere 5 000 nodos. A eso tenemos que sumar 1 dirección más para la interfaz interna del router R1 además de las direcciones de red y broadcast. En total 5 003 direcciones. Ahora debemos calcular su logaritmo en base 2 para determinar cuántos bits de *host-ID* se necesitan:

$$\log_2(5\,003) = 12,28 \quad (7.1)$$

Obviamente no podemos tener bits decimales, de modo que necesitaremos 13 bits para conseguir las 5 003 direcciones. La máscara de esa subred será por tanto $32 - 13 = /19$. Ten en cuenta que con 13 bits tenemos realmente $2^{13} = 8\,192$ direcciones, así que «sobran» $8\,192 - 5\,003 = 3\,189$ direcciones.

Si aplicas estos mismos cálculos al resto de las subredes, obtendrás los resultados que aparecen en el Cuadro 7.4. Para cada red, se muestra el nombre, la cantidad de nodos que necesita (Necesidad), el tamaño en bits del *host-ID*, la máscara la subred resultante (Máscara), la cantidad total de direcciones (Total) y la cantidad de direcciones libres/desperdiciadas (Libres).

Una vez tenemos claras las necesidades podemos proceder al reparto. Por conveniencia se hace siempre el reparto desde la red más grande a la más

Red	Necesidad	host-ID	Máscara	Total	Libres
A	5 003	13	/19	8 192	3 189
B	2 003	11	/21	2 048	45
C	1 003	10	/22	1 024	21
R1-R0	2	2	/30	4	0
R2-R0	2	2	/30	4	0
R3-R0	2	2	/30	4	0

CUADRO 7.4: Ejemplo de VLSM: Necesidades de las subredes

pequeña. Veámoslo paso a paso. Como el bloque de partida es /18 y la red A necesita un bloque /19 debemos utilizar 1 bit. La tabla 7.5 muestra esta división. En la primera columna se muestra el tercer byte de la dirección que es dónde está la frontera entre *net-ID* y *host-id*. El bit marcado en negrita es el que se ha utilizado para dividir la red original.

3 ^{er} byte	Dirección de red	Red
0000 0000	70.50.0.0/19	Red A
0010 0000	70.50.32.0/19	Libre

CUADRO 7.5: Ejemplo de VLSM: Red A

El siguiente paso es asignar espacio para la red B. Como requiere un bloque /21 y tenemos un bloque 70.50.32.0/19, significa que necesitamos solo una cuarta parte, de modo que utilizaremos 2 bits adicionales del *host-id*. La tabla 7.6 muestra la división.

3 ^{er} byte	Dirección de red	Red
0010 0000	70.50.32.0/21	Red B
0010 1000	70.50.40.0/21	Libre
0011 0000	70.50.48.0/21	Libre
0011 1000	70.50.56.0/21	Libre

CUADRO 7.6: Ejemplo de VLSM: Red B

La red C necesita un bloque /22, así que dividimos el primer bloque libre (70.50.40.0/21) con 1 bit adicional. La tabla 7.7 muestra la división.

Por último, utilizaremos el primer bloque libre (70.50.44.0/22) para asignar subredes a los 3 enlaces serie. En este caso se utilizan 8 bits adicionales para obtener máscaras /30. La tabla 7.8 muestra la división. Al realizar la división con 8 bits tendríamos 256 de estos bloques /30, pero como solo necesitamos 3. En la tabla se aparecen únicamente los 4 primeros.

3 ^{er} byte	Dirección de red	Red
0010 1000	70.50.40.0/22	Red C
0010 1100	70.50.44.0/22	Libre

CUADRO 7.7: Ejemplo de VLSM: Red C

3 ^{er} byte	4 ^o byte	Dirección de red	Red
0010 1100	0000 0000	70.50.44.0/30	R1-R0
0010 1100	0000 0100	70.50.44.4/30	R2-R0
0010 1100	0000 1000	70.50.44.8/30	R3-R0
0010 1100	0000 1100	70.50.44.12/30	Libre

CUADRO 7.8: Ejemplo de VLSM: Enlaces serie

Veamos ahora otros datos relevantes de cada bloque. En el Cuadro 7.9 se muestra para cada bloque, la dirección de red, la máscara en decimal, la primera y última dirección asignable y la dirección de broadcast.

Nombre	Dir. red	Máscara	1 ^a dir.	Última dir.	Broadcast
A	70.50.0.0/19	255.255.224.0	70.50.0.1	70.50.31.254	70.50.31.255
B	70.50.32.0/21	255.255.248.0	70.50.32.1	70.50.39.254	70.50.39.255
C	70.50.40.0/22	255.255.252.0	70.50.40.1	70.50.43.354	70.50.43.255
R1-R0	70.50.44.0/30	255.255.255.252	70.50.44.1	70.50.44.2	70.50.44.3
R2-R0	70.50.44.4/30	255.255.255.252	70.50.44.5	70.50.44.6	70.50.44.7
R2-R0	70.50.44.8/30	255.255.255.252	70.50.44.9	70.50.44.10	70.50.44.11

CUADRO 7.9: Ejemplo de VLSM: Resultado

Por último, en el Cuadro 7.10 aparecen todos los bloques que han quedado libres. La primera columna es solo un identificador para referirnos a ellos. Los bloques 1 a 7 corresponden con la fragmentación que hemos causado al asignar los enlaces serie. Hemos agregados en lo posible el espacio disponible para evitar esos 253 bloques /30 que no necesitamos. El bloque 9 también es una agregación de los bloques 70.50.48.0/21 y 70.50.56.0/21 que quedaron libres. Como son contiguos y difieren solo en el primer bit del *net-id* es posible expresarlos como un solo bloque /20.

Id	3 ^{er} byte	4 ^o byte	Dirección del bloque
1	0010 1100	0000 1100	70.50.44.12/30
2	0010 1100	0001 0000	70.50.44.16/28
3	0010 1100	0010 0000	70.50.44.32/27
4	0010 1100	0100 0000	70.50.44.64/26
5	0010 1100	1000 0000	70.50.44.128/25
6	0010 1101	0000 0000	70.50.45.0/24
7	0010 1110	0000 0000	70.50.46.128/23
8	0010 1100	0000 0000	70.50.44.0/22
9	0011 0000	0000 0000	70.50.48.0/20

CUADRO 7.10: Ejemplo de VLSM: Bloques libres

7.5. Direcciones especiales

Además de las direcciones de red y broadcast, hay otras direcciones especiales que tienen usos particulares o predefinidos. Veamos las más importantes:

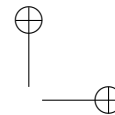
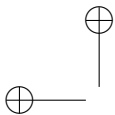
Dirección nula La dirección `0.0.0.0` —conocida como `INADDR_ANY`— se utiliza en varias situaciones diferentes cuando no se conoce, no se dispone o no tiene sentido utilizar una dirección IP concreta. Por ejemplo, en las peticiones ARP, en una tabla de encaminamiento para indicar entrega directa o router por defecto y otros casos que veremos a lo largo del libro.

Direcciones *loopback* Cualquier dirección del rango `127/8` solo tiene sentido en el propio nodo y no se puede asignar a ninguna interfaz de red que pueda ser accesible desde fuera. La dirección más común para este uso es `127.0.0.1`, de la que ya hemos hablado en capítulos anteriores.

Direcciones privadas Son direcciones que se pueden utilizar únicamente en redes privadas, y por ello no necesitan autorización ni asignación por parte de ninguna autoridad, pero son ignoradas por los routers fuera de la red privada. Las veremos en detalle en el capítulo 20.

Direcciones de enlace local Son direcciones que se utilizan para comunicar dispositivos en la misma red local, sin necesidad de un router. Hablaremos de ellas también en el capítulo 20.

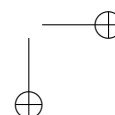
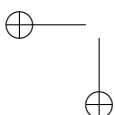
Direcciones multicast Son direcciones que identifican a grupos. No están asignadas a ningún interfaz de red. Sirven para enviar un mensaje a todos los dispositivos que pertenezcan a ese grupo. Se utilizan en aplicaciones como la IPTV, radio por IP, distribución de contenidos, etc. Son las direcciones de clase D y empiezan por `1110`. El uso de

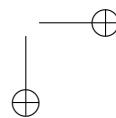
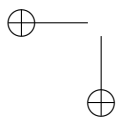
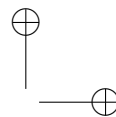
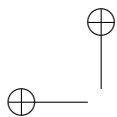


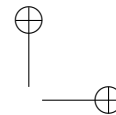
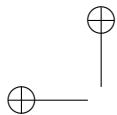
esta clase no fue descartado con la llegada del direccionamiento sin clases, y se sigue utilizando en la actualidad.

Y ¿qué más?

En realidad no hay mucho más que saber sobre el direccionamiento IP. Con lo que has visto en este capítulo ya tienes una buena base para diseñar por ti mismo un esquema de direccionamiento de cualquier organización. El diseño puede ser arbitrariamente complejo y tendrás que considerar muchos factores, pero el proceso de diseño se basa en los conceptos que has visto aquí. Incluso con IPv6 la base es prácticamente la misma en lo que concierne a la subdivisión de bloques, ya que nunca existió direccionamiento con clase en IPv6. En el capítulo 10 veremos cómo se asignan estas direcciones a las interfaces de red, ya sea manual o automáticamente con DHCP, pero esa tarea en realidad es independiente del esquema de direccionamiento.







Capítulo 8

Interconexión

Al terminar este capítulo, entenderás:

- Cómo se mueven los paquetes IP a través de una interred.
- Qué son y qué hacen los routers.
- Qué es una tabla de encaminamiento, qué información contiene y cómo la utiliza.
- Cómo los routers colaboran con otros para llevar los paquetes hasta su destino.
- Cómo pueden los paquetes IP atravesar redes con tecnologías de enlace diferentes y qué es la fragmentación de paquetes.

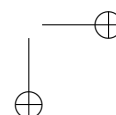
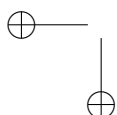
Sin duda alguna, la posibilidad de llevar información desde prácticamente cualquier punto del planeta a cualquier otro es lo que ha hecho de Internet una innovación que ha cambiado para siempre la mayoría de las industrias, el comercio, la economía, el entretenimiento y entre otras muchas cosas, lo más importante, la forma en la que los humanos nos comunicamos.

Una de las tecnologías clave que ha hecho posible esta revolución es la interconexión de redes, particularmente, cuando se trata de tecnologías heterogéneas. Aunque es cierto que existieron y existen otras tecnologías que ofrecen una funcionalidad similar, ha sido TCP/IP la que ha logrado convertirse con diferencia en la más popular.

Por supuesto, llevar paquetes a través de múltiples redes es uno de los objetivos prioritarios de TCP/IP y en particular del protocolo IP. Citando la RFC 791 [9]:

“The purpose of internet protocol is to move datagrams through an interconnected set of networks”.

En español:



“El propósito del protocolo de interred es mover datagramas a través de un conjunto interconectado de redes”.

De nuevo insistimos aquí, por su importancia, que «Protocolo de Internet» debe entenderse como «protocolo de interconexión de redes», porque es IP el que hace posible las interredes.

Para interconectar redes es necesario resolver varios problemas. El principal es el encaminamiento, el mecanismo que determina la ruta que siguen los paquetes hasta su destino. Los routers juegan un papel clave. Son de hecho los dispositivos que crean las interredes. Disponen de varias interfaces, conectadas a redes distintas, y por eso pueden mover paquetes entre ellas. Pero también es necesario resolver los problemas que aparecen cuando se utilizan tecnologías de enlace diferentes al interconectar redes heterogéneas. Y además debemos entender el modo en que los routers informan de errores o situaciones especiales a otros routers y a los nodos finales.

8.1. Almacenamiento y reenvío

Definamos primero lo que es un router: Es un dispositivo conectado a más de una red, que además, tiene la capacidad de aceptar paquetes *dirigidos a terceros* y reenviarlos (del inglés *forwarding*) para colaborar en la tarea de llevarlos hasta su destino.

La diferencia clave entre un PC que actúa como un simple «nodo final» y otro que actúa como router, es que el segundo reenvía los paquetes que no son para él, es decir, aquellos cuya dirección destino no coincide con la dirección asignada a la interfaz por la que llegan. El nodo final en cambio, los descarta.

En un sistema GNU/Linux, ese *reenvío* se activa con el parámetro del núcleo `net.ipv4.ip_forward`:

```
# echo 1 > /proc/sys/net/ipv4/ip_forward
```

Recuerda que para hacer este cambio persistente, debes editar el archivo `/etc/sysctl.conf` (ver § 2.10).

Veamos una topología de ejemplo (Figura 8.1) que vamos a utilizar para ilustrar varios conceptos y mecanismos a lo largo del capítulo. Esta interred está formada por 5 nodos (PC1 a PC5) conectados a 3 redes Ethernet (N1 a N3). Los routers R1 y R2 tiene 2 interfaces Ethernet y una interfaz serie con encapsulación PPP. Para todas las interfaces se indica la dirección IP y el último octeto de la dirección MAC¹.

¹Es solo una simplificación para poder usar aquí datos más sencillos

Aquí podemos ver cómo los routers tienen interfaces en varias redes que además pueden ser de distinta tecnología y protocolo de enlace (Ethernet y PPP en este caso).

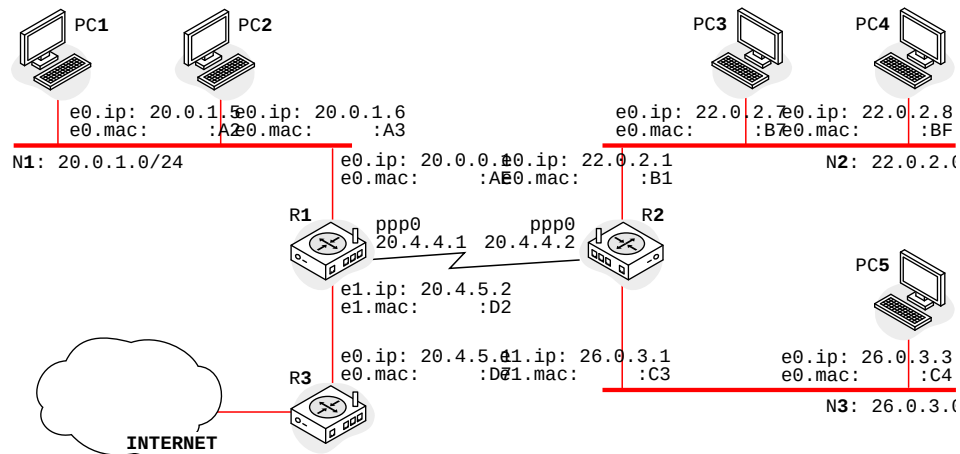


FIGURA 8.1: Topología de ejemplo

IP es una tecnología de conmutación de paquetes, que también se suele llamar «red de datagramas». Un datagrama es un bloque de datos independiente, que lleva incorporada —en su cabecera— toda la información necesaria para llegar a su destino. El router desconoce —o simplemente pasa por alto— si una secuencia de paquetes que está recibiendo pertenecen a la misma conexión o flujo, proceden del mismo origen o van al mismo destino. Aunque tuvieran relación, son tratados como si no la tuvieran. Por eso, paquetes que sí están relacionados podrían seguir rutas diferentes, y eso tiene consecuencias importantes.

Un router IP recibe y procesa los paquetes que van llegando a cualquiera de sus interfaces, sin importar la tecnología de enlace que emplean. Como la lógica del router puede estar ocupada cuando llega un nuevo paquete, es necesario guardar el paquete temporalmente en una cola. De forma similar, cuando el router quiere reenviar un paquete, la línea de salida puede estar ocupada, por lo que también hace falta una cola para almacenar los paquetes pendientes de enviar. Resumiendo, cada interfaz tiene una cola de entrada y otra de salida.

Por esta forma de trabajar se dice que el router es un dispositivo de *almacenamiento y reenvío* (*store and forward*). Veamos con más detalle el proceso que realiza el router cada vez que llega un paquete:

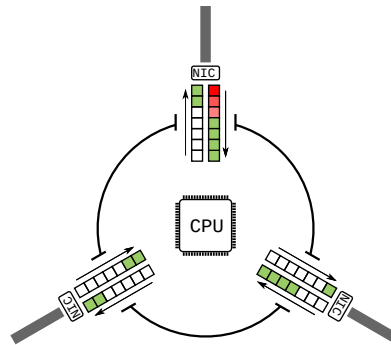


FIGURA 8.2: Esquema de las colas de entrada y salida de un router

1. Una vez recibido por la interfaz, el paquete se extrae de su envoltorio de enlace. La cabecera y cola del protocolo de la trama se descarta. Es una tarea que realiza la propia NIC como en cualquier otro dispositivo.
2. El paquete se almacena en memoria, en concreto, en la cola de entrada de esa interfaz; y espera allí hasta que el router disponga de tiempo/recursos para procesarlo.
3. Cuando por fin el router procesa el paquete, calcula su *checksum* y comprueba que coincida con el que aparece en la cabecera. Si esta comprobación falla, el paquete es descartado (y el origen es informado).
4. Comprueba el campo TTL de la cabecera. Si es 0, se descarta el paquete e informa al origen mediante un mensaje ICMP *time exceeded* (§ 8.6.2 para más detalles).

En otro caso (si es mayor que 0), lo decrementa en 1, calcula el nuevo *checksum* y actualiza ambos campos en la cabecera. El campo TTL evita que un paquete pueda quedar viajando por la red indefinidamente en caso de producirse un bucle de encaminamiento.



El *checksum* al que nos referimos aquí es una función de suma de comprobación que permite detectar cualquier modificación que hayan sufrido los datos durante la transmisión, incluso aunque afecte a un único bit. Se conoce como *Internet checksum* y lo utilizan varios protocolos de la familia TCP/IP. Aparte de la cabecera IPv4 que estamos tratando aquí, lo usan TCP, UDP, ICMP, e IGMP, entre otros. En el repositorio de código del libro puedes encontrar una implementación en Python de este algoritmo ([🔗/inet-checksum/checksum.py](#)) y un notebook con una explicación detallada de su funcionamiento ([🔗/inet-checksum/checksum.ipynb](#)).

5. Determina la interfaz de salida en función exclusivamente de la dirección destino del paquete y del contenido de la tabla de encaminamiento. Si la tabla no contiene información sobre cómo proceder, el paquete se descarta y, de nuevo, informa del problema al origen con un mensaje ICMP *destination unreachable* (destino inalcanzable).
6. El paquete se coloca en la cola de la interfaz de salida elegida.
7. Cuando el medio físico asociado a la interfaz queda libre, la NIC encapsula el paquete IP con el formato de trama según la tecnología de ese enlace que lo transforma en una señal capaz de alcanzar el siguiente dispositivo.

Cuando esta interfaz de salida está conectada a un enlace de difusión (*p. ej.* una LAN Ethernet), el router debe determinar primero la dirección física destino de la trama.

Todo este proceso es lo que llamamos «un salto», una expresión que enfatiza la idea de que el paquete ha pasado (saltado) de una red a otra. Los sucesivos routers a lo largo de la ruta efectúan cada uno de estos saltos hasta el destino. Es la base del funcionamiento cualquier interred IP.

8.2. Entrega directa vs. indirecta

El trabajo del router es entregar el paquete «al siguiente» en la ruta hacia su destino. En este camino, hay dos tipos de entrega:

- La entrega directa (o local) la realiza cuando el destino del paquete es un vecino del origen. Por ejemplo, si ambos (origen y destino) están conectados a una red Ethernet, el origen encapsula el paquete en una trama que lleva en su cabecera la dirección física del destino. Tanto la dirección MAC destino (en la cabecera de la trama) como la IP destino (en la cabecera del paquete) identifican al nodo destino.
- La entrega indirecta ocurre, como puedes suponer, cuando el destino no es vecino y, de hecho, probablemente el origen no tiene ninguna información en absoluto de dónde se encuentra ese destino. En ese caso, el origen debe entregar el paquete a un router asumiendo que él sabe cómo llevarlo al destino. Aquí también se encapsula el paquete en una trama, pero la dirección física destino es la del siguiente router, no la del destino, es decir, la trama va dirigida al router, pero el paquete encapsulado va dirigido al destino.

Fíjate que aunque en esta explicación sirve tanto para routers como para nodos finales. Ambos, cuando necesitan enviar un paquete a un vecino, realizan una entrega directa, y cuando el destino está fuera de la red, realizan una entrega indirecta.

En la sección 8.5 veremos un ejemplo práctico con datos y mensajes concretos que ilustra ambos tipos de entrega en una interred sencilla, pero no trivial.

8.3. Tabla de encaminamiento

La tabla de encaminamiento contiene la información que necesita el router para hacer su trabajo principal: decidir dónde enviar cada paquete que llega. Cada fila de esta tabla especifica un destino (normalmente una red) y qué hacer para llegar a él. Sin embargo, la tabla no tiene instrucciones precisas de cómo llevar el paquete hasta allí, solo habla del siguiente salto. Veamos un ejemplo en la Tabla 8.1.

#	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 8.1: Ejemplo de tabla de encaminamiento

Vale la pena hacer aquí una aclaración para evitar malentendidos. La tabla de encaminamiento (*routing table*) se denomina muy a menudo en español «tabla de rutas». Pero si entendemos que una «ruta» es un «itinerario o camino para un viaje»², la traducción «tabla de rutas» es incorrecta, o al menos, confusa. Como acabamos de explicar, este tipo de tabla por norma general no contiene rutas. Por eso optaremos aquí por el término «tabla de encaminamiento». A pesar de esto, en muchas ocasiones (incluso en inglés) para hacer referencia a las filas de la tabla, se habla de «rutas».

Toda tabla de encaminamiento contiene al menos la siguiente información:

Destino (*dst*)

Una dirección IP que identifica una red. Aunque menos frecuente, también puede ser una dirección de nodo. Un caso especial es la dirección IP nula (0.0.0.0) que indica que esa fila es un *router por defecto*.

Máscara (*mask*)

La máscara asociada al destino (la primera columna) y determina qué parte de aquella corresponde al prefijo de red (ver sección 7.2). En ocasiones la columna «destino» se combina con la máscara en formato CIDR (*p. ej.* 120.10.12.0/24).

²Así lo define la RAE.

Siguiente salto (*next-hop*)

La dirección IP de un router vecino, o bien la dirección IP nula (0.0.0.0) que aquí significa entrega directa.

Interfaz (*iface*)

El nombre de la interfaz por la que debe salir el paquete.

Para cada nuevo paquete que llega, al router busca la correspondencia entre la dirección destino del paquete y la combinación destino-máscara de cada fila. Como varias filas podrían cumplir esta correspondencia, se aplica el siguiente criterio de prioridad:

1. Entrega directa.
2. Entrega indirecta.
3. Router por defecto.

Dentro de cada categoría se aplica LPM (Longest Prefix Match), es decir, la máscara más larga (más específica) tiene más prioridad. Si aún así hay empate, se aplica la primera fila que aparezca en la tabla.

Si no es posible encontrar ninguna coincidencia, el paquete es descartado (y el origen informado con un mensaje ICMP). Sin embargo, la tabla puede incluir *routers por defecto* (*default routers*). Normalmente solo hay uno y suele aparecer en la última fila (como en la Tabla 8.1), pero eso es solo una convención y en realidad la posición dónde aparezca es irrelevante.

Pero exactamente, ¿cuál es esa «correspondencia» que se evalúa para cada fila? Consiste simplemente en una operación AND a nivel de bits entre la dirección destino del paquete y la máscara de esa fila. La fila es adecuada si el resultado de esa operación coincide con el valor de la columna «destino».

Cuando el router determina qué fila utilizar, envía el paquete a la dirección que aparece en la columna «siguiente salto» a través de la interfaz indicada en la columna «interfaz». Fíjate que la columna «interfaz» siempre identifica interfaces del propio router.

Siguiendo el ejemplo, supongamos que al router de la Tabla 8.1 llega un paquete con destino 40.0.0.9. La única fila coincidente sería la , lo cual se determina al aplicar la operación indicada:

$$(40.0.0.9 \text{ AND } 255.255.255.0) == 40.0.0.0$$

Por tanto, el paquete se reenviará al destino, que es un vecino (es una entrega directa) a través de la interfaz e1. El procesamiento del paquete termina y el router procede con el siguiente paquete.

Veamos un fragmento de pseudocódigo que ilustra el modo en que el router procesa la tabla de encaminamiento por cada paquete que llega a través de cualquiera de sus interfaces:

```
destino = datagrama.ip_destino
for fila in tabla_de_encaminamiento:
    if (destino AND fila.máscara) == fila.destino:
        reenviar(datagrama, fila.siguiente_salto)
        return

if router_por_defecto:
    reenviar(datagrama, router_por_defecto)
else:
    enviar_ICMP(destino_inalcanzable, datagrama.origen)
```

8.3.1. Analizando la tabla de ejemplo

Estudiemos en detalle la Tabla 8.1 que corresponde con un hipotético router **R0** que podría ser perfectamente realista. Veamos la información que ofrece. La primera columna se ha añadido únicamente para numerar las filas y referirnos a ellas.

- La tabla corresponde a un router que dispone de 3 interfaces: **e0**, **e1** y **s0**. Aunque cada fabricante y SO sigue su propia nomenclatura para nombrar las interfaces, las Ethernet tienen nombres como **e0** o **eth1** y las interfaces serie **s0** o bien con el protocolo de enlace que utilizan, como **ppp1**.
- 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 si el destino está en la red **30.0.0.0/24** (accesible por medio de su interfaz **e0**) o en la red **40.0.0.0/24** (a través de su interfaz **e1**), el router y el destino son vecinos y el propio router se encarga de entregar el paquete.
- Las filas 3 y 4 especifican entregas indirectas. La 3 indica que todo paquete dirigido a la red **130.10.20.0/24** debe reenviarse al router **30.0.0.2**. La fila 4 que todo paquete dirigido a la red **120.20.30.0/17** debe reenviarse al encaminador **40.0.0.3**.
- Por último, la fila 5 establece un router por defecto. Cualquier paquete que no haya satisfecho ninguna de las otras filas, será enviado al router **50.1.1.2**.

Es importante señalar que la tabla de encaminamiento no dice nada sobre dónde están las redes **130.10.20.0/24** y **120.20.30.0/17** ni a cuántos saltos se encuentran. La tabla solo dice que para llegar a las redes *distantes*, se debe delegar la entrega al router vecino indicado. Pero hay parte de la

topología que podemos deducir (ver Figura 8.3), la que involucra a las redes *locales*, es decir, de las que R0 es vecino.

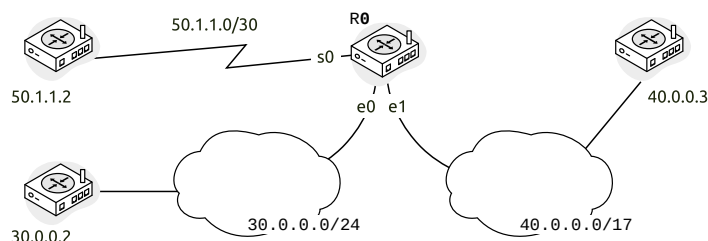


FIGURA 8.3: Topología (parcial) que se deduce de la Tabla 8.1

8.3.2. Tabla de nodo final

Cualquier nodo o dispositivo conectado a Internet (incluso un *smart phone* o una consola de videojuegos) tiene una tabla de encaminamiento, aunque eso no lo convierte en router. Como vimos, para comportarse como router debe ser capaces de hacer reenvío (ver § 8.1).

La tabla de encaminamiento indica al nodo final cómo enviar tráfico a sus vecinos y cómo enviar tráfico fuera del enlace (una LAN normalmente). Una tabla de encaminamiento típica de un nodo en una red doméstica con enlace WiFi³ será muy parecida 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

¿Qué significa esta tabla?

- La primera fila dice dos cosas: quiénes son los vecinos (los conectados a la red 192.168.0.0/24), y que para llegar a ellos el nodo debe hacer una entrega directa (*next-hop* = 0.0.0.0).
- La segunda fila indica el router por defecto, que identifica un router de la red domestica (192.168.0.1); 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, puedes ver la tabla de encaminamiento con el comando `ip route`. Como probablemente tu computador es un PC conectado a una red doméstica, verás algo muy similar a lo siguiente:

³Suele estar detrás de un router NAT. Más adelante veremos qué implica eso.

```
$ ip route
default via 192.168.8.1 dev wlp1s0 proto dhcp metric 600
192.168.0.0/24 dev wlp1s0 proto kernel scope link metric 600
```

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	wlp1s0
192.168.0.0/24	scope link	wlp1s0

Que se corresponde directamente a esa tabla básica de la que hablábamos justo antes. Las diferencias más significativas son:

- Las columnas «destino» y «máscara» aparecen como un solo dato en notación CIDR: *destino/máscara*.
- Como destino *nulo*, en lugar de 0.0.0.0, muestra *default* (por *default router*).
- Para el siguiente salto, en lugar de 0.0.0.0 muestra *scope link*, que significa que esos dispositivos (192.168.0.0/24) son accesibles en el «ámbito del enlace», es decir, son vecinos.

La designación *proto dhcp* indica que esa entrada se ha creado a partir de información recibida por medio del servicio DHCP, mientras que *proto kernel* indica que ha sido creada por el SO, probablemente cuando se activó la interfaz.

Con *metric* se indica la prioridad. Si hay varias filas aplicables para un mismo paquete, se elige la que tenga menor valor de prioridad. Si la prioridad también coincide, en el caso de GNU/Linux, el SO aplica un algoritmo de balanceo de carga llamado ECMP que puede tener en cuenta cosas como la carga de la interfaz o la caché (prioriza la última usada). Un router dedicado, normalmente implementado como un ASIC o FPGA puede utilizar los criterios que considere el fabricante o su configuración específica.

8.4. Agregación

En ocasiones una tabla de encaminamiento puede contener filas que especifican redes que comparten un prefijo y son contiguas. En estos casos, es posible realizar *agregación* (*summarization*) de esos destinos. Por ejemplo, supongamos la topología de la Figura 8.4.

Siendo la tabla de encaminamiento de R1:

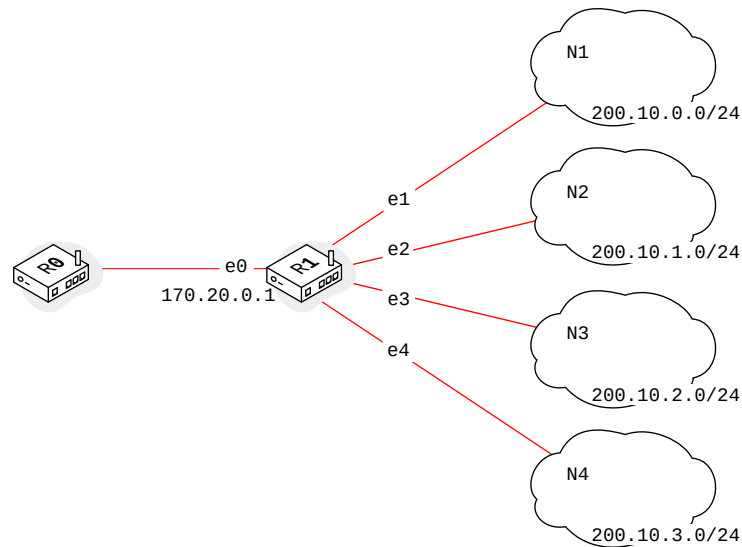


FIGURA 8.4: Topología con redes agregables

dst/mask	next hop	iface
200.10.0.0/24	0.0.0.0	e0
200.10.1.0/24	0.0.0.0	e1
200.10.2.0/24	0.0.0.0	e2
200.10.3.0/24	0.0.0.0	e3
170.20.0.0/24	0.0.0.0	e4

La tabla de R0 en principio debería incluir también una fila para llegar a cada una de esas redes:

dst/mask	next hop	iface
200.10.0.0/24	170.20.0.1	e0
200.10.1.0/24	170.20.0.1	e0
200.10.2.0/24	170.20.0.1	e0
200.10.3.0/24	170.20.0.1	e0
170.20.0.0/24	0.0.0.0	e0

Pero una mejor opción sería «agregar» esas 4 filas en una sola, es decir, puede indicar que para llevar el paquete a una hipotética red 200.10.0.0/22 se debe enviar a R1. La tabla de R0 quedaría así:

dst/mask	next hop	iface
200.10.0.0/22	170.20.0.1	e0
170.20.0.0/24	0.0.0.0	e0

Esto reduce el tamaño de la tabla de **R0** (y de todos a los que llegue este destino) sin perder información. Conlleva un aumento de rendimiento y una mejora de la escalabilidad de la red. Lo podemos ver como una especie de *supernetting* con la diferencia de que esta red agregada no existe en ningún sentido, salvo a efectos de encaminamiento.

8.5. Un par de ejemplos detallados

Sobre la topología de ejemplo (Figura 8.1) veamos cómo funciona el encaminamiento con un par de escenarios concretos. Analizamos con todo detalle el proceso que sigue un paquete IP desde su creación en el origen hasta alcanzar su destino, incluyendo los cambios en su encapsulación a lo largo de la ruta.

Obviamente hace falta conocer las tablas de encaminamiento de routers y nodos. Las de **R1** y **R2** aparecen en la Figura 8.2.

dst/mask	next hop	iface	dst/mask	next hop	iface
20.0.1.0/24	0.0.0.0	e0	22.0.2.0/24	0.0.0.0	e0
22.0.2.0/24	20.4.4.2	ppp0	26.0.3.0/24	0.0.0.0	e1
26.0.3.0/24	20.4.4.2	ppp0	0.0.0.0/0	20.4.4.1	ppp0
20.4.4.2/32	0.0.0.0	ppp0	20.4.4.1/32	0.0.0.0	ppp0
20.4.5.0/30	0.0.0.0	e1			
0.0.0.0	20.4.5.1	e1			

CUADRO 8.2: Tablas de encaminamiento de **R1** y **R2**

Las demás tablas de encaminamiento son tablas básicas típicas que suelen tener los nodos finales como la que hemos visto en § 8.3.2. Por ejemplo, la tabla de **PC1** sería algo como:

dst/mask	next hop	iface
20.0.1.0/24	0.0.0.0	e0
0.0.0.0	20.0.1.1	e0

8.5.1. Escenario 1: entrega local

Supongamos que **PC1** envía un paquete IP a **PC2**, por ejemplo, ejecutando `ping -c1 20.0.1.6`. El proceso implica los siguientes pasos:

1. **PC1**⁴ construye el mensaje ICMP y lo encapsula en un paquete IP.

⁴Nombramos los nodos, pero lógicamente estas tareas las realizan sus SO.

IP	ver+IHL 0x45 tos 0 len 0x0054 id 0x1C46 flags+offset 0x0000 ttl 64 proto 1 (ICMP) cksum src 20.0.1.5 dst 20.0.1.6
ICMP	type 8 (Echo Request) code 0 cksum id 0x1234 seq 1

2. PC1 consulta su tabla de encaminamiento. La primera fila es aplicable (20.0.1.6 AND 255.255.255.0 == 20.0.1.0), de modo que realiza entrega directa.
3. Como la interfaz de salida es Ethernet, PC1 debe encapsular el paquete en una trama Ethernet, pero desconoce la dirección MAC de PC2.
4. Para averiguarla, envía una petición ARP a la dirección broadcast, por lo que llegará a todos los nodos de la red. La petición ARP pregunta «¿Quién tiene la dirección IP 20.0.1.6?». La cabecera ARP incluye esencialmente lo siguiente:
 - IP origen: 20.0.1.5, IP destino: 20.0.1.6.
 - MAC origen: :A2, MAC destino: 00:00:00:00:00:00.

El mensaje completo sería:

Ethernet	dst FF:FF:FF:FF:FF:FF src :A2 type 0x0806 (ARP)
ARP	hw 0x0002 (eth) proto 0x0800 (IP) hw-len 6 proto-len 4 op 0x0001 sender :A2 20.0.1.5 target 00:00:00:00:00:00 20.0.1.6

5. PC2 recibe la petición ARP y responde con su dirección MAC (:A3) en una respuesta ARP.

Ethernet	dst :A2 src :A3 type 0x0806 (ARP)
ARP	hw 0x0002 proto 0x0800 hw-len 6 proto-len 4 op 0x0002 sender :A3 20.0.1.6 target :A2 20.0.1.5

6. PC1 puede ahora construir y enviar el paquete IP encapsulado en otra trama Ethernet.

Ethernet	dst :A3 src :A2 type 0x0800 (IP)
IP	ver+IHL 0x45 tos 0 len 0054 id 0x1C46 flags+offset 0x0000 ttl 64 proto 1 (ICMP) cksum src 20.0.1.5 dst 20.0.1.6
ICMP	type 8 (Echo Request) code 0 cksum id 0x1234 seq 1

Como ya dice el título de la sección, se trata de una entrega **directa** dado que PC1 y PC2 son vecinos, pero en todo caso a la vista del mensaje es fácil de comprobar. En una entrega directa ambas direc-

ciones destino, tanto física (:A3) como lógica (20.0.1.6) corresponden al nodo destino (PC2).

7. La trama llega a PC2, que la desencapsula y procesa el paquete IP.
8. Para responder al mensaje ping, PC2 crea una respuesta *Echo Reply* y la envía a PC1 aplicando el mismo procedimiento que se acaba de describir, solo que a la inversa.

IP	ver+IHL 0x45 tos 0 len 0054 id 0x1C46 flags+offset 0x0000 ttl 64 proto 1 (ICMP) cksum src 20.0.1.6 dst 20.0.1.5
ICMP	type 0 (Echo Reply) code 0 cksum id 0x1234 seq 1

8.5.2. Escenario 2: dos saltos

Probemos algo un poco más interesante. Supongamos que PC1 ejecuta ahora `ping -c1 26.0.3.3`. En este caso, el paquete debe atravesar 2 routers y 3 redes hasta PC5.

1. PC1 construye el paquete IP con las direcciones correspondiente:

IP	ver+IHL 0x45 tos 0 len 0054 id 2D57 flags+offset 0x0000 ttl 64 proto 1 (ICMP) cksum src 20.0.1.5 dst 26.0.3.3
ICMP	type 8 (Echo Request) code 0 cksum id 0x5678 seq 1

2. PC1 consulta su tabla de encaminamiento. Ninguna fila convencional es aplicable a la dirección destino, pero hay un router por defecto (20.0.1.1), así que realiza una entrega indirecta hacia él.
3. Como antes, necesita averiguar la dirección MAC de la interfaz *e0* del router, para lo cual envía una petición ARP con las siguientes direcciones:
 - IP origen: 20.0.1.5, IP destino: 20.0.1.1 (R1).
 - MAC origen: :A2, MAC destino: 00:00:00:00:00:00.
4. R1 responde con la dirección MAC (:AE) en una respuesta ARP.
5. PC1 puede ahora construir y enviar el paquete IP encapsulado en otra trama Ethernet.

Ethernet	dst :AE src :A2 type 0x0800 (IP)
IP	ver+IHL 0x45 tos 0 len 0054 id 2D57 flags+offset 0x0000 ttl 64 proto 1 (ICMP) cksum src 20.0.1.5 dst 26.0.3.3
ICMP	type 8 (Echo Request) code 0 cksum id 0x5678 seq 1

En este caso se puede comprobar que la dirección física destino (:AE) corresponde a R1 mientras que la lógica (26.0.3.3) corresponde a PC5, es decir, se trata de una entrega **indirecta**.

6. La trama llega a R1, que la desencapsula y procesa el paquete IP.
7. Como el paquete no es para él (26.0.3.3 $\neq$ 20.0.1.1) procede a reenviarlo. Consulta su tabla de encaminamiento. La única fila aplicable es la tercera (26.0.3.3 AND 255.255.255.0 == 26.0.3.0) de modo que realiza una nueva entrega indirecta a R2.
8. El enlace R1-R2 es serie y utiliza una encapsulación PPP. En este caso no se utiliza ARP porque no hay otro destino posible que el otro extremo del enlace. Tampoco hay direcciones MAC en este tipo de interfaz. La trama PPP será algo como:

PPP	addr 0xFF control 0x03 proto 0x0021 (IP)
IP	ver+IHL 0x45 tos:0 len 0054 id 2D57 flags+offset 0x0000 ttl 64 proto 1 (ICMP) cksun src 20.0.1.5 dst 26.0.3.3
ICMP	type 8 code 0 cksun id 0x5678 seq 1

9. R2 recibe la trama PPP, la desencapsula y procesa el paquete IP. Tampoco es para él (26.0.3.3 $\neq$ 20.4.4.2), así que lo reenvía. La segunda fila de su tabla de encaminamiento es aplicable, e indica una entrega directa por la interfaz e1.
10. R2 tiene que averiguar la dirección MAC del destino. Envía una petición ARP preguntando por 26.0.3.3 y PC5 responde con su dirección: :C4.
11. R2 puede ahora construir y enviar el paquete IP en una trama Ethernet con las siguientes direcciones:
 - MAC origen: :C3, MAC destino: :C4.
12. PC5 recibe la trama, desencapsula el paquete IP y a su vez el mensaje ICMP. Como es un *Echo request*, procede a construir una respuesta *Echo reply* dirigida a PC1, que recorrerá el camino en sentido contrario.

8.6. Mensajes de control de interred (ICMP)

Ya hemos hablado de ICMP en § 5.7, y lo hemos estado utilizando a menudo con el comando `ping`, que utiliza uno de los tipos de mensajes ICMP, y en este mismo capítulo hemos visto otros usos. En esta sección veremos con más detalle los distintos mensajes y su finalidad [11]. Primero recordemos el formato general del mensaje:

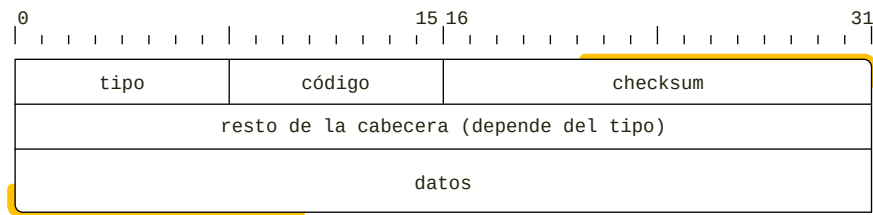


FIGURA 8.5: Formato del mensaje ICMP

Hay dos grupos de mensajes ICMP: los de notificación de errores y los de consulta/diagnóstico. El campo **tipo** indica el tipo de mensaje mientras que el campo **código** indica un subtipo. Veremos a continuación los más comunes.

Los mensajes de notificación de errores son enviados cuando un router detecta un problema en el encaminamiento o entrega de un paquete⁵. Estos mensajes son enviados siempre al nodo origen del paquete que ha causado el problema. Es importante entender que el objetivo de estos mensajes es meramente informativo. IP es un protocolo esencialmente no confiable e ICMP no pretende cambiar eso. Del mismo modo que no hay garantías de que un paquete IP pueda ser entregado, tampoco las hay para un mensaje ICMP, de hecho los mensajes ICMP se encapsulan sobre paquetes IP. Algunos de los mensajes de notificación son los siguientes (se muestra entre corchetes el valor del campo **tipo**):

- Destino inalcanzable [3]
- Tiempo excedido [11]
- Problema en parámetro [12]
- Supresión al origen [4]
- Redirección [5]

Los mensajes ICMP de error incluyen como carga útil al menos la cabecera IP del paquete problemático y los primeros 8 bytes de la carga útil de aquel, lo que normalmente corresponderá a los puertos origen y destino de un mensaje TCP o UDP que este estuviera transportando en el momento de producirse el error. Esta información es muy útil para que el SO del nodo origen determine qué proceso es el responsable y puede informar al programador mediante un error o excepción, para que él pueda identificar y tratar el problema.

Nunca se generan mensajes ICMP cuando el paquete problemático porta un mensaje ICMP de notificación de error. Esto es porque informar de un

⁵Ya hemos visto algunos casos en este mismo capítulo.

problema de entrega podría generar a su vez otro problema de entrega, provocando un bucle. Como ICMP Echo-request no es un mensaje de error, por regla general sí se envían mensajes ICMP de error si hay problemas al entregarlo.

Los mensajes de consulta/diagnóstico, como su nombre indica, tratan de averiguar cierta información o comprobar el estado. Por eso estos mensajes vienen por pares petición-respuesta. Son los siguientes:

- Ping (petición [8]/respuesta [0])
- Marca de tiempo (petición [13]/respuesta [14])
- Información (petición [15]/respuesta [16])
- Máscara de subred (petición [17]/respuesta [18])
- Router (solicitud [10]/anunciamiento [9])

Estudiémoslos uno a uno.

8.6.1. Destino inalcanzable (*Destination unreachable*)

Es una notificación muy frecuente. Un router envía un mensaje *Destino inalcanzable* (tipo=3) cuando no puede entregar un paquete a su destino. Hay varios motivos por lo que eso puede ocurrir, y corresponden con los distintos valores para el campo código:

0. Red inalcanzable (*network unreachable*). Lo utiliza el router si no encuentra una fila en su tabla de encaminamiento que pueda utilizar para reenviar el paquete.
1. Nodo inalcanzable (*host unreachable*). El router tiene acceso a la red destino, pero no puede efectuar la entrega directa.
2. Protocolo inalcanzable (*protocol unreachable*). El destino no soporta el protocolo encapsulado en el paquete IP.
3. Puerto inalcanzable (*port unreachable*). El puerto destino en el nodo destino está cerrado.
4. Fragmentación necesaria (*fragmentation needed and DF set*). El paquete necesita ser fragmentado, pero el origen ha solicitado que no se fragmente (ver § 8.7).
5. Fallo en la ruta en origen (*source route failed*). No se ha podido aplicar la opción *source route* de IP⁶.

Es sencillo reproducir algunos de estos errores porque tu propio sistema los genera en caso de problemas cuando intenta enviar tráfico. Puedes provocar un mensaje *nodo inalcanzable* enviando ping a una dirección IP que sabes que no está asignada a ningún nodo de la red. En el siguiente ejemplo

⁶ *Source route* es una función de diagnóstico de IP con un uso muy marginal.

asumimos que la IP de tu nodo es 192.168.0.37/24 y la del nodo que no existe es 192.168.0.66/24. Cambia estas direcciones por las adecuadas en tu caso y pruébalo por ti mismo.

```
$ ping -c 1 192.168.0.66
PING 192.168.0.66 (192.168.0.66) 56(84) bytes of data.
From 192.168.0.37 icmp_seq=1 Destination Host Unreachable
```

Si capturas el tráfico verás el mensaje ICMP en detalle. Como siempre eliminamos los campos irrelevantes para el propósito que nos ocupa:

```
1 $ sudo tshark -i any -f icmp -V
2 Internet Protocol Version 4, Src: 192.168.0.37, Dst: 192.168.0.37
3   Total Length: 112
4   Protocol: ICMP (1)
5   Source Address: 192.168.0.37
6   Destination Address: 192.168.0.37
7 Internet Control Message Protocol
8   Type: 3 (Destination unreachable)
9   Code: 1 (Host unreachable)
10  Internet Protocol Version 4, Src: 192.168.0.37, Dst: 192.168.0.66
11    Total Length: 84
12    Protocol: ICMP (1)
13    Source Address: 192.168.0.37
14    Destination Address: 192.168.0.66
15 Internet Control Message Protocol
16   Type: 8 (Echo (ping) request)
17   Code: 0
```

Fíjate que el mensaje ICMP se lo envía el sistema a sí mismo (192.168.0.37) porque ha sido el propio sistema el que ha detectado el problema. También puedes ver el mensaje ICMP Echo-request (con su cabecera IP) encapsulado en el mensaje de error (**líneas 10-17**).

El programador puede acceder a esta información desde el código. Así pues, un intento de conexión a un nodo inexistente eleva una excepción `socket.error` como consecuencia directa de que el SO reciba y procese el mensaje ICMP. Puedes verlo en el Listado 8.1. Al ejecutar este programa verás un mensaje similar a «[Errno 113] No route to host».

```
import socket

try:
    sock = socket.socket()
    sock.connect(('192.168.0.66', 1234))
except socket.error as e:
    print(e)
```

LISTADO 8.1: Captura del error «nodo inalcanzable» con Python

8.6.2. Tiempo excedido (*Time exceeded*)

Recuerda que los routers decrementan el campo TTL cuando procesan un paquete IP. Pues bien, si el valor de TTL llega a cero, el router descarta el paquete y envía un mensaje ICMP de este tipo con `código=0`, que significa formalmente *tiempo de vida excedido en tránsito* (*time to live exceeded in transit*).

Los nombres tanto del error como del campo de la cabecera IP *time to live* (tiempo de vida) al que está asociado, son una reminiscencia de la Internet primigenia cuando los routers tardaban aproximadamente un segundo en procesar cada paquete. El campo TTL expresaba (en segundos) la vida que aún le quedaba al paquete. Hoy en día hay routers que pueden procesar millones de paquetes por segundo y el significado práctico del campo TTL ahora es el número de saltos que el paquete tiene permitido dar aún. En esta línea y para ser coherente con el significado actual, en el diseño de IPv6 el campo equivalente pasó a llamarse *hop-count* (cuenta de saltos).

Si el campo `código=1`, el mensaje lo envía el nodo destino para indicar que ha agotado el tiempo límite en espera de recibir todos los fragmentos de un paquete. En este caso significa *tiempo excedido en el reensamblado* (*time to live exceeded in reassembly*). El valor recomendado para ese tiempo límite es de 60 segundos [9], aunque por ejemplo en GNU/Linux es configurable mediante el parámetro del núcleo `net/ipv4/ipfrag_time`.

traceroute

El programa `traceroute` aprovecha el mecanismo que acabamos de describir para descubrir la ruta que sigue un paquete IP hacia su destino. `traceroute` comienza enviando tres mensajes ICMP *Echo*, o un segmento UDP o TCP encapsulado en un paquete IP con `TTL=1`. Eso provoca que el router local descarte el paquete, informe del error y con ello revele su dirección IP (porque aparece como dirección origen del mensaje de error). A continuación, `traceroute` repite el envío con `TTL=2`, obteniendo la dirección del segundo router en la ruta, y así sucesivamente hasta alcanzar el nodo destino.

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 al router local de esa LAN (161.67.101.1). Al lado aparecen los tiempos RTT para cada uno de los tres paquetes enviados. Para cada router aparece primero su nombre (si tiene) y después su dirección IP. Como se puede apreciar la ruta completa requiere 6 saltos, es decir, el paquete atraviesa 6 routers. La última entrada es el destino solicitado.

8.6.3. Problema en parámetro (*Parameter problem*)

Este mensaje de error lo envía un router o un nodo destino cuando detecta un valor incorrecto en la cabecera IP que le impide procesarla. El mensaje incluye un campo puntero que indica el campo de la cabecera en el que ha encontrado el problema.

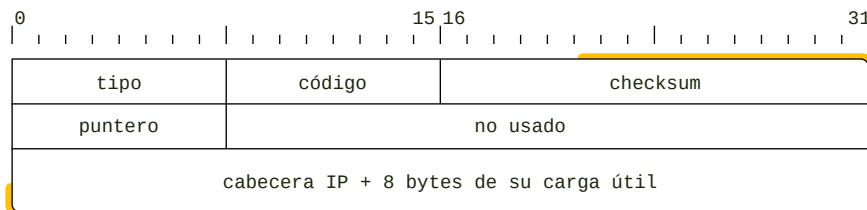


FIGURA 8.6: Mensaje ICMP de problema en parámetro

8.6.4. Supresión al origen (*Source quench*)

Este mensaje notifica un problema de exceso de tráfico, que puede estar relacionado con un problema de congestión si lo envía un router o como mecanismo de control de flujo si lo envía el destino. En todo caso, este mecanismo está obsoleto, no se implementa en sistemas modernos y no se recomienda su uso [15]. Trataremos este tema con más detalle en § 13.4.

8.6.5. Redirección (*Redirect*)

Este mensaje lo envía un router para indicar a un nodo origen que hay una ruta mejor para llegar a su destino. El mensaje incluye la dirección IP del router alternativo. Este mensaje no es técnicamente un error, es más bien informativo. En todo caso, el uso de este mensaje también está obsoleto porque podría ser explotado por un atacante malicioso en una técnica MITM.

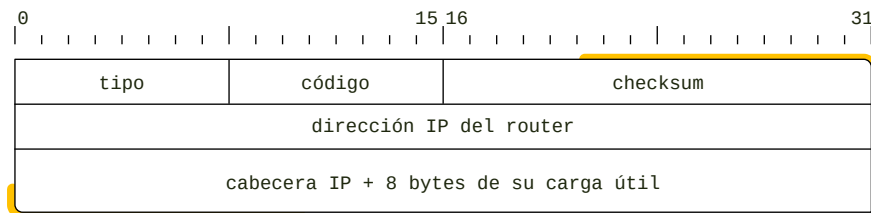


FIGURA 8.7: Mensaje ICMP de redirección

Existe un parámetro del núcleo `net/ipv4/conf/all/accept_redirects` que permite configurar el comportamiento de los mensajes de redirección (por defecto se ignoran).

8.6.6. Ping (*Echo*)

Ya hemos hablado bastante de `ping` y lo hemos utilizado en varias ocasiones, incluso hemos visto algunas capturas (ver §5.7) de los mensajes ICMP que genera. El programa `ping` envía un mensaje ECHO request (`tipo=8`, `código=0`) a un nodo remoto. Éste al recibirlo envía de vuelta un mensaje ECHO reply (`tipo=0`, `código=0`).

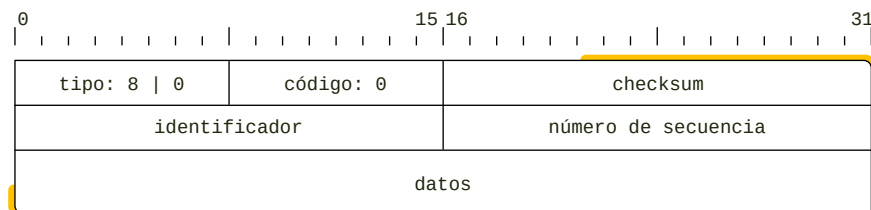


FIGURA 8.8: Mensaje ICMP Echo

El mensaje tiene un campo `identificador` y un campo `número de secuencia`. En una determinada ejecución del programa `ping` todos los mensajes de petición utilizan el mismo identificador y cada mensaje lleva un número de secuencia creciente empezando en 1. Aquí puedes ver una captura del comando `ping example.net`:

```
$ sudo tshark -f icmp
Capturing on 'eno1'
1 192.168.0.37 → 104.18.5.106 ICMP 98 Echo (ping) request id=0x0002, seq=1/256, ttl=64
2 104.18.5.106 → 192.168.0.37 ICMP 98 Echo (ping) reply id=0x0002, seq=1/256, ttl=57
3 192.168.0.37 → 104.18.5.106 ICMP 98 Echo (ping) request id=0x0002, seq=2/512, ttl=64
4 104.18.5.106 → 192.168.0.37 ICMP 98 Echo (ping) reply id=0x0002, seq=2/512, ttl=57
5 192.168.0.37 → 104.18.5.106 ICMP 98 Echo (ping) request id=0x0002, seq=3/768, ttl=64
6 104.18.5.106 → 192.168.0.37 ICMP 98 Echo (ping) reply id=0x0002, seq=3/768, ttl=57
```

Ese campo *id* hace posible dos o más ejecuciones de *ping* con el mismo origen y destino. Recuerda que ICMP se encapsula directamente sobre IP, aquí no hay puertos que permitan multiplexar⁷. El segundo número tras '/' en el campo *seq* es el mismo valor, pero expresado en *little endian*.

La carga útil del mensaje de petición tiene por defecto una longitud de 56 bytes. Los primeros 16 bytes suelen ser una marca de tiempo y después 40 bytes de datos aleatorios o consecutivos. Puedes comprobarlo con una captura:

```
$ sudo tshark -i lo -f icmp -V
Internet Control Message Protocol
  Type: 8 (Echo (ping) request)
  Code: 0
  Timestamp from icmp data: Apr 21, 2025 18:38:03.966021000 CEST
  Data (40 bytes)

0000  10 11 12 13 14 15 16 17 18 19 1a 1b 1c 1d 1e 1f  .....
0010  20 21 22 23 24 25 26 27 28 29 2a 2b 2c 2d 2e 2f  !"#$$%&'()*+,-./
0020  30 31 32 33 34 35 36 37                          01234567

Data: 101112131415161718191a1b1c1d1e1f202122232425262728292a2b2c2d2e2f3031323334353637
```

Realmente no hay un campo *timestamp* en el mensaje ICMP, pero es tan habitual que la carga útil sea un *timestamp*, que *tshark* así lo interpreta. Puedes ver que los siguientes bytes empiezan con el valor `0x10` y van creciendo a partir de ahí hasta `0x37`.

El mensaje Echo reply que envía el receptor debe enviar de vuelta la misma carga útil que le envió el emisor —por eso se llama *eco*. El emisor puede entonces tomar la hora de llegada y restar la que él mismo envió en el mensaje de petición. Con ello obtiene el lapso de tiempo que ha transcurrido entre la petición y la respuesta. Ese es el tiempo de ida y vuelta o RTT (Round-Trip Time), que muestra la salida del programa con la etiqueta *time* el imprimir cada respuesta.

```
~$ ping -c2 8.8.8.8
PING 8.8.8.8 (8.8.8.8) 56(84) bytes of data.
64 bytes from 8.8.8.8: icmp_seq=1 ttl=117 time=4.25 ms
64 bytes from 8.8.8.8: icmp_seq=2 ttl=117 time=4.27 ms

--- 8.8.8.8 ping statistics ---
2 packets transmitted, 2 received, 0% packet loss, time 1001ms
rtt min/avg/max/mdev = 4.481/4.526/4.571/0.045 ms
```

⁷distinguir entre procesos

8.6.7. Marca de tiempo (*Timestamp*)

El emisor envía su marca de tiempo en el mensaje de petición (tipo=13). El receptor envía un mensaje de respuesta (tipo=14) con tres marcas de tiempo:

- origen: La marca de tiempo que contenía el mensaje de petición.
- recepción: El instante en que llegó el mensaje de petición.
- transmisión: El instante en que envió el mensaje de respuesta.

Como el mensaje Echo, incluye un campo *identificador* y *número de secuencia* para que el emisor pueda realizar la correspondencia entre petición y respuesta.

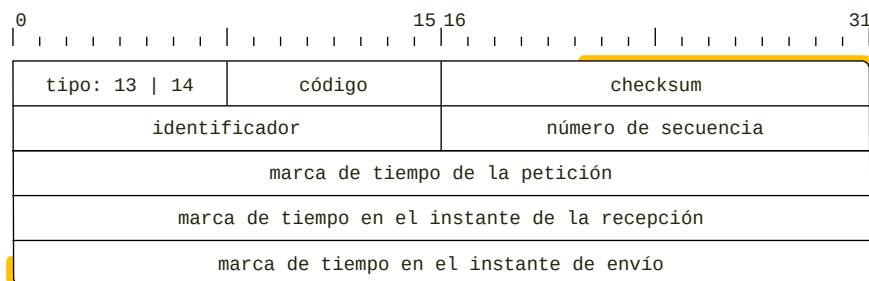


FIGURA 8.9: Mensaje ICMP Timestamp

Puedes enviar un mensaje de marca de tiempo con el programa `hping3`, incluido en el paquete del mismo nombre. A continuación puedes ver un ejemplo:

```
~$ sudo hping3 -c 4 --icmp --icmp-ts example.net
HPING example.net (eno1 23.215.0.141): icmp mode set, 28 headers + 0 data bytes
len=46 ip=23.215.0.141 ttl=50 id=42928 icmp_seq=0 rtt=91.9 ms
ICMP timestamp: Originate=63921874 Receive=63921915 Transmit=63921915
ICMP timestamp RTT tsrtt=92

len=46 ip=23.215.0.141 ttl=50 id=43150 icmp_seq=1 rtt=91.8 ms
ICMP timestamp: Originate=63922874 Receive=63922915 Transmit=63922915
ICMP timestamp RTT tsrtt=92

len=46 ip=23.215.0.141 ttl=50 id=43869 icmp_seq=2 rtt=91.7 ms
ICMP timestamp: Originate=63923875 Receive=63923915 Transmit=63923915
ICMP timestamp RTT tsrtt=91

len=46 ip=23.215.0.141 ttl=50 id=44269 icmp_seq=3 rtt=91.6 ms
ICMP timestamp: Originate=63924875 Receive=63924915 Transmit=63924915
ICMP timestamp RTT tsrtt=91

--- example.net hping statistic ---
4 packets transmitted, 4 packets received, 0% packet loss
```

```
round-trip min/avg/max = 91.6/91.7/91.9 ms
```

Lo que vemos a la salida son los mensajes de respuesta, que incluyen las tres marcas: *Originate*, *Receive* y *Transmit*. Las marcas *Receive* y *Transmit* son idénticas porque el tiempo que transcurre entre la recepción de la petición y el envío de la respuesta es despreciable.

El mensaje ICMP Timestamp permite calcular un RTT preciso sólo si los relojes de ambos nodos están sincronizados o el desfase (sesgo) es conocido. Como esto es poco habitual, un uso que puede ser interesante es precisamente para realizar una sincronización o medir el sesgo. El emisor puede combinar el mensaje Echo para estimar y corregir el sesgo de su reloj, y de ese modo sincronizar su reloj con el del destino.

8.6.8. Información (*Information*) y máscara (*Address mask*)

Los mensajes de petición/respuesta de información (tipos 15 y 16), y petición/respuesta de máscara (tipos 17 y 18) constituían un mecanismo rudimentario para que un nodo pudiera averiguar la dirección IP o máscara asignada a ese nodo. Están obsoletos [16] y no se utilizan en la actualidad. El protocolo DHCP suple con creces esta necesidad.

8.6.9. Solicitud y anuncio de router

Estos mensajes forman parte del protocolo IRDP (ICMP Router Discovery Protocol) [17], y como sus nombres indican, sirven respectivamente para que un nodo pueda solicitar routers locales para poder salir de la LAN y para que un router pueda anunciar, ya sea como respuesta al mensaje de solicitud o de forma proactiva para informar de su presencia.

Como podrás suponer, es una funcionalidad en desuso ya que el protocolo DHCP cubre esta necesidad con creces. Puedes ver información detallada sobre configuración de nodos con DHCP en § 10.2.

8.7. Fragmentación

La función principal del router es interconectar redes, y en muchas situaciones esas redes pueden ser heterogéneas, es decir, estar basadas en tecnologías de enlace diferentes, como el caso de la Figura 8.1.

Uno de los problemas que surge al interconectar redes heterogéneas es la disparidad en los tamaños de trama. En concreto el problema aparece cuando se extrae la carga útil que llega en la trama entrante y se necesita encapsular en una trama de un tamaño menor para ser enviada por una interfaz diferente. Al tamaño máximo (en bytes) que puede transportar una trama

(la carga útil) de una determinada tecnología se le llama MTU (Maximum Transmission Unit).

La MTU es una característica básica inherente a cualquier interfaz de red y tecnología de enlace, es decir, todas las interfaces de una determinada tecnología tienen siempre la misma MTU en cualquier lugar o instalación; es una característica dada por diseño y fabricación. En GNU/Linux lo puedes consultar con el comando `ip link show`. En el siguiente ejemplo puedes ver que la interfaz `eno1` (una NIC Ethernet) tiene una MTU de 1 500 bytes.

```
$ ip link show eno1
2: eno1: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc fq_codel state UP mode DEFAULT
    group default qlen 1000
    link/ether f8:5f:2a:c0:ff:ee brd ff:ff:ff:ff:ff:ff
```

Aunque es muy poco habitual y a pesar de lo dicho, el administrador puede cambiar la MTU de una interfaz. Salvo que haya una buena razón no es nada recomendable y desde luego nunca para configurar valores mayores que la MTU real del enlace. Provocaría todo tipo de problemas y errores difíciles de detectar.

```
$ sudo ip link set dev eth0 mtu 1400
```

La discrepancia entre MTUs puede ocurrir, por ejemplo, cuando un paquete IP que viaja encapsulado en una trama Ethernet o WiFi (MTU=1 500 bytes) llega a un router que lo debe enviar por un enlace serie (MTU=256 bytes). La solución a este problema es que el router **fragmente** (trocee) la carga útil del paquete IP original y cree nuevos paquetes IP (fragmentos) que quepan en las tramas del enlace de salida. A este proceso es a lo que se llama «fragmentación».

El problema no termina ahí. En su viaje hacía el destino, los fragmentos podrían tener que atravesar otras redes con MTU aún menores, con lo que el router correspondiente debería dividirlos a su vez en fragmentos más pequeños. Aunque la probabilidad es baja, podría ocurrir además que los fragmentos, que son a todos los efectos paquetes IP individuales, sigan eventualmente rutas distintas. Eso significa que, para un mismo flujo, podrían llegar al destino paquetes completos y fragmentos de distintos tamaños dependiendo de la ruta que ha seguido cada uno.

Por estos dos motivos: fragmentaciones sucesivas y rutas distintas, el «re-emsamblado», es decir, la unión de los fragmentos para obtener de nuevo el paquete original, solo lo puede realizar el destinatario. Es el único lugar dónde es seguro que van a pasar todos los fragmentos.

La fragmentación es una tarea costosa que consume tiempo y recursos tanto en los routers como en el destino. El destino debe almacenar en memoria los fragmentos que va recibiendo hasta que llegue el último de ellos, y esto puede estar ocurriendo para varios datagramas en varios flujos al mismo tiempo. En muchas ocasiones es preferible evitar la fragmentación. Eso es posible si el nodo origen crea paquetes lo suficientemente pequeños como para quepan en la menor MTU de la ruta probable, que llamamos PMTU (Path MTU).

La cabecera del paquete IP tiene 4 campos relacionados con la fragmentación. Todos ellos están en la segunda palabra de 32 bits, es decir, la segunda fila tal como se suele representar.

versión	IHL	tipo de servicio	longitud total			
identificación			DF	MF	offset	
TTL		protocolo	checksum			
dirección origen						
dirección destino						

FIGURA 8.10: Campos relacionados con la fragmentación en la cabecera IP

Veamos su significado con más detalle:

- **identificación** (*identification*): es un número único que identifica el paquete original. Todos los fragmentos creados a partir de un mismo paquete comparten el mismo número de identificación. De este modo, el destinatario puede determinar qué fragmentos corresponden con qué paquete original.
- **Flag DF**. Si está activo, indica que el paquete no debe ser fragmentado: DF (Don't Fragment).
- **Flag MF**. Si está activo significa que ese paquete no es el último fragmento, hay más: MF (More Fragments).
- **offset**: Es un número que indica la posición que ocupa la carga útil de ese fragmento respecto al paquete original. El primer fragmento siempre tiene un **offset**=0. Fíjate que la longitud de este campo es de solo 13 bits, mientras que el tamaño del paquete IP se expresa con un entero de 16 bits. Eso es porque este campo está expresado en múltiplos de 8 bytes, lo que lógicamente implica que los fragmentos no pueden tener cualquier tamaño, tiene que ser múltiplo de 8 bytes.

Con esto, se pueden distinguir cuatro situaciones al inspeccionar una cabecera IP:

1. Si `offset=0` y el flag MF no está activo, se trata de un paquete completo no fragmentado.
2. Si `offset=0` y el flag MF está activo, el paquete ha sido fragmentado y este es el primer fragmento.
3. Si `offset` es distinto de 0 y el flag MF está activo, el paquete ha sido fragmentado y este es un fragmento intermedio.
4. Si `offset` es distinto de 0 y el flag MF no está activo, el paquete ha sido fragmentado y este es el último fragmento.

Fíjate que cuando un paquete está fragmentado no es posible saber cuántos fragmentos lo componen. El destino los irá recibiendo y almacenando temporalmente, pero no sabrá cuantos son hasta que rellene todos los huecos.

Veamos un ejemplo de fragmentación con la topología de la Figura 8.11.

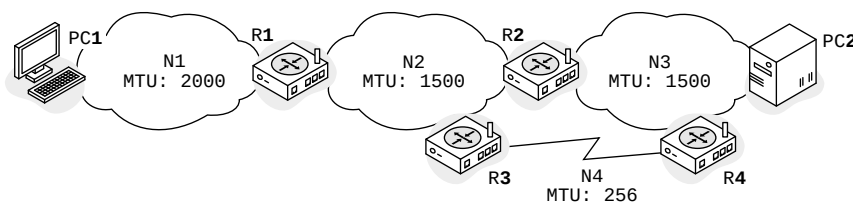


FIGURA 8.11: Topología para el ejemplo de fragmentación

Supongamos que **PC1** envía un paquete IP a **PC5** con una carga útil de 1 980 bytes, lo que implica un tamaño total de 2 000 bytes considerando que incluye una cabecera IP estándar sin opciones (20 bytes). Por regla general (como este caso) todo nodo evita crear paquetes mayores que la MTU de la red a la que está conectado, pues eso le obligaría a fragmentar ya desde el origen.

El valor para los campos relevantes sería el siguiente cuando el paquete sale a la red **N1**:

- Tamaño total: 2 000 bytes.
- identificación: 0x1234 (ejemplo).
- Flag DF: 0 (fragmentación permitida).
- Flag MF: 0 (es el último fragmento).
- `offset`: 0 (es el primer fragmento).
- Tamaño de la carga útil: 1 980 bytes.
- Rango de datos: 0-1 979.

Es el primer y último fragmento, es decir, no está fragmentado (el primer caso de la lista anterior). Al llegar a **R1**, éste debe fragmentarlo para poder

enviarlo a la red N2 que requiere una encapsulación con MTU=1 500 bytes, lo que implica que la carga útil máxima es de 1 480 bytes (1 500 - 20). Crea por tanto dos fragmentos:

- *Fragmento 0*
 - Tamaño total: 1 500 bytes.
 - identificación: 0x1234.
 - Flag MF: 1.
 - offset: 0.
 - Tamaño de la carga útil: 1 480 bytes.
 - Rango de datos: 0–1 479.
- *Fragmento 1:*
 - Tamaño total: 520 bytes.
 - identificación: 0x1234.
 - Flag MF: 0.
 - offset: 185 (1 480 / 8).
 - Tamaño de la carga útil: 500 bytes.
 - Rango de datos: 1 480–1 979.

Ahora supongamos que R1 encamina el fragmento 0 por R2, que está conectado a otra red con MTU=1500 (N3), es decir, no requiere más fragmentación y llega tal cual al destino. Sin embargo, R1 encamina el fragmento 1 por R3 y este a su vez hacia R4 hasta el destino. Eso obliga a R3 a fragmentar de nuevo pues el tamaño de este segundo fragmento (520 bytes) supera el MTU del enlace serie N4 (256 bytes). Restando la cabecera, la carga útil del paquete podría tener un máximo de 236 bytes (256 - 20). Pero 236 no es múltiplo de 8, así que el primer fragmento deberá tener 232 bytes de carga útil.

Los campos relevantes de estos nuevos fragmentos serían:

- *Fragmento 1.1:*
 - Tamaño total: 252 bytes.
 - identificación: 0x1234.
 - Flag MF: 1.
 - offset: 185 (1 480 / 8).
 - Tamaño de la carga útil: 232 bytes.
 - Rango de datos: 1 480–1 711.
- *Fragmento 1.2:*
 - Tamaño total: 252 bytes.
 - identificación: 0x1234.
 - Flag MF: 1.
 - offset: 232 (1 712 / 8).
 - Tamaño de la carga útil: 232 bytes.
 - Rango de datos: 1 712–1 943.
- *Fragmento 1.3:*
 - Tamaño total: 56

- identificación: 0x1234.
- Flag MF: 0.
- offset: 243 (1944 / 8).
- Tamaño de la carga útil: 36 bytes.
- Rango de datos: 1943–1979.

La Figura 8.12 representa las cabeceras de los fragmentos que se han creado. En la primera fila aparece el tamaño total del paquete (cabecera + carga útil), en la segunda fila los campos relacionados con la fragmentación: **identificador**, **flag DF** y **offset**, y en la parte inferior la carga útil, en la que se representa un rango que numera los bytes empezando en 0.

Como hemos visto, al entrar el paquete original en la red N2 se crean los fragmentos 0 y 1, y posteriormente cuando el fragmento 1 entra en la red N4 se crean a partir de él los fragmentos 1.1, 1.2 y 1.3. Por tanto al destino llegan 4 fragmentos: 0, 1.1, 1.2 y 1.3. Puedes comprobar que los rangos de datos representados en la carga útil son contiguos y coinciden con el paquete original.

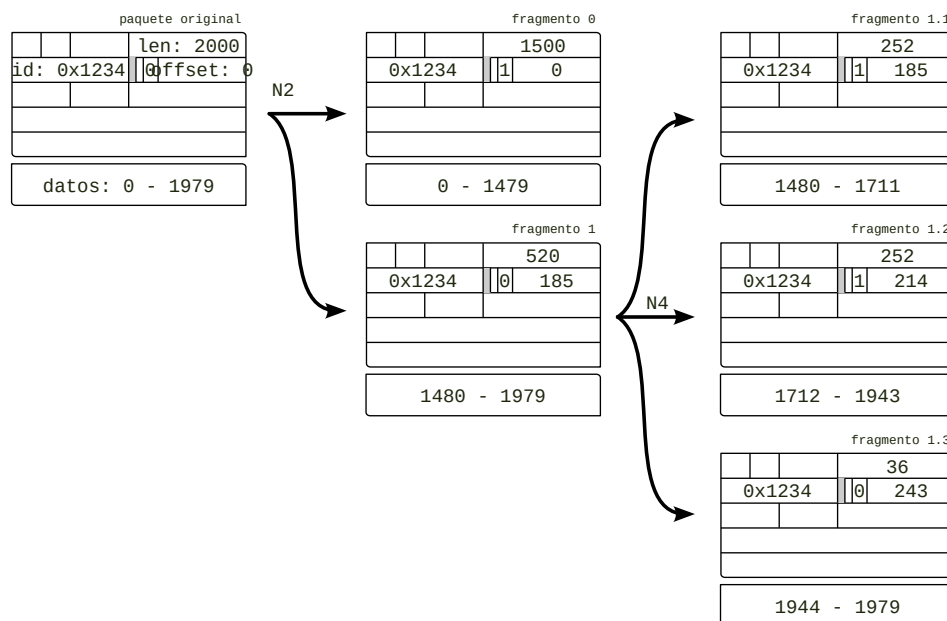


FIGURA 8.12: Ejemplo de fragmentación

La fragmentación también agrava la sobrecarga de cabeceras porque crea paquetes IP adicionales. Hemos pasado de un solo paquete IP a cuatro, es decir, 3 nuevas cabeceras IP, además de las cabeceras de las correspondien-

tes tramas adicionales. Si por ejemplo esas tramas fuesen Ethernet, esta sobrecarga se puede cuantificar en $3 \times (14 + 20) = 102$ bytes.

Llevemos esto a la práctica con un pequeño laboratorio llamado `lab-frag` basado en `docker`. Para ello, vamos a recrear la topología trivial de la Figura 8.13

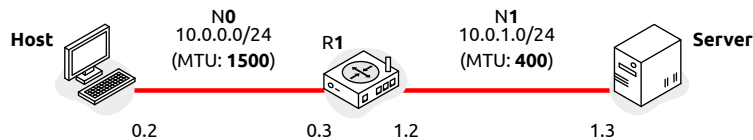


FIGURA 8.13: Topología del laboratorio de la fragmentación `lab-frag`

Esta topología está especificada en el archivo `compose.yml` para `docker-compose`.

```
x-common: &common-settings
  privileged: true
  restart: "no"
  ulimits:
    nofile:
      soft: 100000
      hard: 200000

services:
  router:
    build: ./router
    image: lab-frag-router
    container_name: router
    hostname: router
    <<: *common-settings
    networks:
      N0: { ipv4_address: 10.0.0.3 }
      N1: { ipv4_address: 10.0.1.2 }

  server:
    build: ./server
    image: lab-frag-server
    container_name: server
    <<: *common-settings
    networks:
      N1: { ipv4_address: 10.0.1.3 }

networks:
  N0:
    ipam: { config: [{subnet: 10.0.0.0/24}] }
  N1:
    driver_opts: { com.docker.network.driver.mtu: 400 }
    ipam: { config: [{subnet: 10.0.1.0/24}] }
```

LISTADO 8.2: Descripción del laboratorio de fragmentación `lab-frag/compose.yml`

Con este tenemos un router y un nodo **Server** en la red **N1** que tiene una MTU de 400 bytes. El nodo **host** es tu computador real. Lo único que debes tener en cuenta es que previamente a la ejecución del ejemplo tu dirección de red no esté dentro de **10.0.0.0/8** ni tengas nada en tu tabla de rutas que lleve a esa red. De otro modo tendrías un conflicto y no funcionaría.

En el directorio **lab-frag** ejecuta **make** para arrancar el laboratorio. Una vez en marcha puedes ejecutar **ping** y **traceroute** para comprobar que puedes llegar al nodo **Server**.

```
$ ping -c1 10.0.1.3
PING 10.0.1.3 (10.0.1.3) 56(84) bytes of data.
64 bytes from 10.0.1.3: icmp_seq=1 ttl=63 time=0.049 ms

--- 10.0.1.3 ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 0.049/0.049/0.049/0.000 ms
$ traceroute
traceroute to 10.0.1.3 (10.0.1.3), 30 hops max, 60 byte packets
 1 10.0.0.3 (10.0.0.3) 0.494 ms 0.409 ms 0.372 ms
 2 10.0.1.3 (10.0.1.3) 0.349 ms 0.316 ms 0.275 ms
```

Este primer *ping* es un mensaje muy pequeño (64 bytes) y no se ha visto afectado por la fragmentación. Ahora puedes enviar un mensaje mucho mayor (700 bytes de carga útil) con **ping -c1 -s700 10.0.1.3** para forzar a **R1** a fragmentar.

Si ejecutas **tshark** en **Server** y vuelves a ejecutar **ping** podrás ver los fragmentos.

```
$ docker exec server tshark -f icmp
Capturing on 'eth0'
1 10.0.0.1 ? 10.0.1.3 IPv4 410 Fragmented IP protocol (proto=ICMP 1, off=0, ID=073c)
2 10.0.0.1 ? 10.0.1.3 ICMP 366 Echo (ping) request id=0x0008, seq=1/256, ttl=63
3 10.0.1.3 ? 10.0.0.1 IPv4 410 Fragmented IP protocol (proto=ICMP 1, off=0, ID=d41e)
4 10.0.1.3 ? 10.0.0.1 ICMP 366 Echo (ping) reply id=0x0008, seq=1/256, ttl=64
```

Los mensajes 1 y 2 corresponden a la petición ICMP Echo. En el primer mensaje **tshark** ofrece información del fragmento (**offset** y **identificación**) porque en ese momento obviamente no tiene toda la información. Al llegar el segundo fragmento ya puede reensamblar y determina que es un mensaje Echo. Los mensajes 3 y 4 son la respuesta que envía **Server**, que necesariamente tiene que ser del mismo tamaño, y por esa razón el propio nodo se ve obligado a fragmentar *en origen*.

Puedes hacer una captura más detallada para ver el valor de los campos de la cabecera IP. Mostramos aquí solo la información relevante de las cabeceras IP de los 2 primeros mensajes (el Echo request).

```
$ docker exec server tshark -f icmp -V
Frame 1:
Internet Protocol Version 4, Src: 10.0.0.1, Dst: 10.0.1.3
  Total Length: 396
  Identification: 0x85b8 (34232)
  001. .... = Flags: 0x1, More fragments
  ...0 0000 0000 0000 = Fragment Offset: 0
  Source Address: 10.0.0.1
  Destination Address: 10.0.1.3

Frame 2:
Internet Protocol Version 4, Src: 10.0.0.1, Dst: 10.0.1.3
  Total Length: 352
  Identification: 0x85b8 (34232)
  000. .... = Flags: 0x0
  ...0 0000 0010 1111 = Fragment Offset: 376
  Source Address: 10.0.0.1
  Destination Address: 10.0.1.3
  [2 IPv4 Fragments (708 bytes): 1(376), 2(332)]
    [Frame: 1, payload: 0-375 (376 bytes)]
    [Frame: 2, payload: 376-707 (332 bytes)]
    [Fragment count: 2]
    [Reassembled IPv4 length: 708]
    [Reassembled IPv4 data: [...]]
```

Es interesante señalar que el tamaño del primer fragmento es 376 y no 400, ya que 380 (400-20) no es múltiplo de 8. Fíjate en estos campos:

- Mensaje 1
 - longitud total: 396 bytes (376 + 20).
 - identificación: 0x85b8.
 - Flag MF: 1.
 - offset: 0.
- Mensaje 2
 - longitud total: 352 bytes (332 + 20).
 - identificación: 0x85b8.
 - Flag MF: 0.
 - offset: 101111=47 (376/8).

Además en el último fragmento tshark muestra información detallada del reensamblado. Indica el tamaño total del paquete original (700) y el de los fragmentos y el rango de los datos de ambos fragmentos.

```
[2 IPv4 Fragments (708 bytes): 1(376), 2(332)]
[Frame: 1, payload: 0-375 (376 bytes)]
[Frame: 2, payload: 376-707 (332 bytes)]
```

Puedes probar a enviar *ping* de otros tamaños o cambiar la MTU en el archivo `compose.yml`, y comprobar los resultados capturados en cada caso.

8.7.1. No fragmentar

Existen algunas situaciones en las que el origen necesita evitar la fragmentación, para lo cual marca el flag DF. Algunas de estas situaciones son:

- El destino es un dispositivo empujado con capacidades muy limitadas y sin capacidad para reensamblar fragmentos.
- La ruta atraviesa una VPN o túnel. La fragmentación rompería la encapsulación.
- El flujo utiliza algún protocolo de tiempo real como RTP o VoIP en el que la fragmentación introduciría latencia y jitter muy perjudicial.

Cuando un router recibe un paquete marcado con el bit DF, descarta el paquete y envía un mensaje ICMP de tipo *Destination unreachable* y código *Fragmentation Needed* al nodo origen. El mensaje de error indica el tamaño máximo de la carga útil que puede aceptar la interfaz de salida de ese router.

Es sencillo provocar esta situación con el comando ping con los argumentos `-M do`, que fija el flag DF, y `-s`, que indica el tamaño de la carga útil. Para el caso de una interfaz Ethernet o Wi-Fi fijamos una carga útil de 1472 bytes, que sumando la cabecera ICMP (8 bytes) y la cabecera IP (20 bytes) da un total de 1500 bytes, que es la MTU de esa tecnología. El comando por tanto sería:

```
$ ping -c 1 -M do -s 1472 8.8.8.8
PING 8.8.8.8 (8.8.8.8) 1472(1500) bytes of data.
From 172.20.21.254 icmp_seq=1 Frag needed and DF set (mtu = 1370)
```

Si pones en marcha una captura antes de ejecutar el comando anterior recibirás algo similar a lo siguiente. Se omiten los campos poco relevantes:

```
$ sudo tshark -f "icmp" -V
Internet Protocol Version 4, Src: 172.20.21.254, Dst: 192.168.0.37
  Total Length: 576
  Time to Live: 62
  Protocol: ICMP (1)
  Source Address: 172.20.21.254
  Destination Address: 192.168.0.37
Internet Control Message Protocol
  Type: 3 (Destination unreachable)
  Code: 4 (Fragmentation needed)
  MTU of next hop: 1370
Internet Protocol Version 4, Src: 192.168.0.37, Dst: 8.8.8.8
  Total Length: 1500
  Identification: 0x0000 (0)
  010. .... = Flags: 0x2, Don't fragment
  ...0 0000 0000 0000 = Fragment Offset: 0
  Time to Live: 62
  Protocol: ICMP (1)
```

```
Source Address: 192.168.0.37
Destination Address: 8.8.8.8
Internet Control Message Protocol
Type: 8 (Echo (ping) request)
Code: 0
[...]
```

Fíjate que el mensaje ICMP que informa del error incluye la cabecera IP y parte de la carga útil (el mensaje ping) que provocó el problema. También puedes comprobar que el mensaje de error lo está enviando 172.20.21.254, que es el router que no ha podido reenviar el paquete. En concreto indica que el enlace saliente al que trataba de entregar el paquete tiene una MTU de 1370 bytes.

8.7.2. Descubrimiento de MTU

El mensaje ICMP «*fragmentation needed*» (§8.6.1) se puede utilizar para descubrir el mayor valor de MTU que permite la ruta hasta un determinado destino. El origen puede empezar enviando un paquete *ping* con un tamaño igual a la MTU de su interfaz de salida y el flag DF activa. Si recibe un mensaje ICMP de este tipo, envía un nuevo mensaje del tamaño indicado en el error. Así sucesivamente hasta recibir la respuesta a *ping* por parte del destino. Este mecanismo se denomina PMTUD (Path MTU Discovery). Indirectamente, también determina el valor de MSS que se utilizará en las conexiones TCP.

Es posible consultar o modificar si el SO aplica PMTUD con el parámetro del núcleo `ip_no_pmtu_disc`. Esa abreviatura se puede leer como «no IP PMTU discovery», es decir, un valor de 0 significa que el descubrimiento está habilitado, que es la situación por defecto.

```
$ cat /proc/sys/net/ipv4/ip_no_pmtu_disc
0
```

Como algunos routers o incluso el destino podrían no procesar mensajes ICMP, existe una estrategia diferente basada en TCP llamada PLPMTUD (Packetization Layer Path MTU Discovery) o «*TCP MTU probing*». También es posible elegir qué estrategia PMTUD se ha de utilizar con la variable `tcp_mtu_probing`.

```
$ cat /proc/sys/net/ipv4/tcp_mtu_probing
0
```

Esta variable puede tomar 3 valores:

- 0: Utiliza siempre la estrategia basada en ICMP (PMTUD clásico).

- 1: Inicialmente utiliza la estrategia basada en ICMP, pero si no recibe respuestas, cambia a PLPMTUD.
- 2: Utiliza siempre PLPMTUD.

Por último, el socket también permite configurar la fragmentación y el comportamiento de PMTUD para un flujo específico. Se realiza por medio de la opción `IP_MTU_DISCOVER`. Puede tomar 6 valores con los siguientes significados:

- `IP_PMTUDISC_DONT` (0): No realizar descubrimiento de PMTU y permitir la fragmentación.
- `IP_PMTUDISC_WANT` (1): Intenta PMTUD clásico. Si no funciona, permite la fragmentación.
- `IP_PMTUDISC_PROBE` (2): Aplicar PMTUD clásico, nunca fragmentar.
- `IP_PMTUDISC_DO` (3): Aplicar PLPMTUD.
- `IP_PMTUDISC_INTERFACE` (4): Utilizar la MTU de la interfaz.
- `IP_PMTUDISC_OMIT` (5): Como la anterior, pero permite fragmentación si el tamaño excede la MTU de la interfaz.

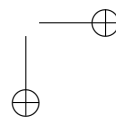
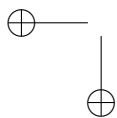
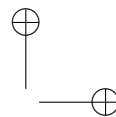
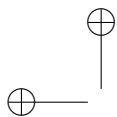
Por ejemplo, puedes utilizar el siguiente código:

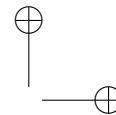
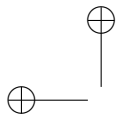
```
import socket
IP_MTU_DISCOVER = 10
IP_PMTUDISC_DO = 2

sock = socket.socket()
sock.setsockopt(socket.IPPROTO_IP, IP_MTU_DISCOVER, IP_PMTUDISC_DO)
```

Y ¿qué más?

En este capítulo hemos abordado una de las funciones clave de cualquier interred IP: llevar paquetes desde un origen a un destino remoto atravesando múltiples redes con tecnologías heterogéneas y probablemente incompatibles. Aquí se ve el papel *homogeneizador* del protocolo IP y la importancia de los routers como elementos de interconexión. Hemos visto también el protocolo ICMP y el problema que supone la disparidad de tamaños de trama propios de cada tecnología de enlace. En el capítulo 9 complementaremos toda esta funcionalidad estudiando como una interred puede descubrir automáticamente las rutas óptimas a cada posible destino gracias a los algoritmos y protocolos de encaminamiento dinámico.





Capítulo 9

Encaminamiento dinámico

Al terminar este capítulo, entenderás:

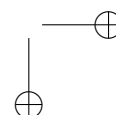
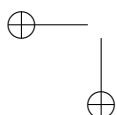
- La necesidad y el funcionamiento del encaminamiento dinámico.
- Qué son los sistemas autónomos y su importancia en el funcionamiento de Internet.
- Cómo funcionan los algoritmos de encaminamiento vector-distancia, estado de enlace y vector ruta.
- Las nociones básicas de los protocolos RIP y OSPF.

Cuando la interred es muy pequeña, las tablas de encaminamiento las crea y mantiene manualmente el administrador. Tanto si las calcula a mano como si utiliza algún programa de vez en cuando, llamamos a eso «encaminamiento estático». Cuando se produce algún cambio, el administrador es el responsable de detectarlo y adaptar las tablas.

Estos cambios pueden deberse a múltiples causas: puede ser un nuevo router (o uno que desaparece), un enlace que falla, una ruta mejor para llegar a un destino ya conocido, un cambio en la forma en la que se calcula la bondad de una ruta o una decisión puramente administrativa.

En cuanto la interred crece un poco, y esos cambios empiecen a producirse de forma habitual, mantener las tablas manualmente se convierte en una tarea compleja y propensa a errores. Se necesita una forma de detectar cambios en la topología y en las condiciones de la red, y algoritmos que puedan adaptar las tablas de encaminamiento inmediatamente y de acuerdo a esos cambios. En eso consiste el encaminamiento dinámico y su principal característica es que es **adaptativo**, es decir, detecta los cambios en la red y modifica las tablas de encaminamiento para adecuarse a la nueva situación.

Por contra, con encaminamiento estático, las tablas no cambian automáticamente, no hay nada que detecte los cambios, calcule nuevas rutas ni modifique las tablas.



La tabla de encaminamiento no es diferente en un caso u otro. De hecho es posible y habitual que una misma tabla contenga filas especificadas de forma estática o manual con otras obtenidas por un protocolo de encaminamiento dinámico, o incluso varios simultáneamente.

9.1. Algoritmos y protocolos de encaminamiento

Un algoritmo de encaminamiento es un algoritmo distribuido utilizado por un grupo de routers que colaboran para deducir la topología de la subred que los conecta. Con la información que es capaz de recabar por sí mismo y la que los otros le pueden proporcionar, cada router calcula la mejor ruta para llegar a cada uno de los destinos y, a partir de eso, actualiza su tabla de encaminamiento.

Los routers colaboran compartiendo información sobre ellos mismos y también sobre lo que han aprendido. Obviamente esta información viaja dentro de mensajes, por lo que, asociados a los algoritmos, existen también protocolos de encaminamiento, con sus correspondientes formatos de mensaje y patrones de intercambio.

El conjunto de routers que ejecutan el algoritmo de encaminamiento funciona a todos los efectos como un sistema distribuido, formado por un conjunto de procesos que se comunican mediante paso de mensajes. En concreto se trata de un algoritmo de *consenso* distribuido, es decir, un algoritmo que permite a un grupo de procesos llegar a un acuerdo sobre el valor de una variable: la topología de la red, y eso a pesar de que algunos de ellos puedan fallar o no responder. Tampoco hay ningún elemento central, árbitro o coordinador que tenga un papel relevante. Aunque por razones de eficiencia o escalabilidad algunos protocolos eligen un router para algún rol especial, este mecanismo también es distribuido y permite elegir un sustituto si deja de responder.

En general, un protocolo de encaminamiento es un conjunto de reglas que define cómo los routers intercambian información sobre la topología de la red y cómo utilizan esa información para calcular las rutas óptimas. Pero del uso del protocolo deriva una limitación importante: cierto grado de obsolescencia de la información.

En los sistemas distribuidos, se dice que «no existe un reloj global», es decir, que es físicamente imposible sincronizar las acciones de un sistema formado por nodos que se comunican mediante mensajes. Este es exactamente el caso de los algoritmos de encaminamiento dinámico. No hay forma de que los routers envíen (y mucho menos reciban) la información topológica exactamente en el mismo instante. Esto implica que los routers trabajan siempre

con datos eventualmente obsoletos. A determinado nivel de precisión (quizá centésimas o milésimas de segundo) tampoco es posible saber cómo de actuales son los datos recibidos, ni las diferencias entre mensajes recibidos de orígenes diferentes. Esto debe considerarse en el diseño de los algoritmos y protocolos para que resulten robustos a pesar de esta limitación.

9.2. Sistemas autónomos

Antes de continuar debemos abordar el concepto de *sistema autónomo* (AS). Un AS es un conjunto de routers y enlaces que dependen o pertenecen a una misma entidad administrativa.

Es habitual que grandes empresas, gobiernos, universidades, proveedores de acceso a Internet (ISP) tengan su propio AS. Algunas de estas empresas pueden tener varios, como es el caso de Google o Amazon.

Cada AS se identifica con un número único de 16 bits (ASN) que, como es habitual, es asignado por la IANA y gestionado por los RIR. Con el crecimiento de Internet en 2007 se crearon los ASN de 32 bits. Actualmente ambos tipos de identificadores conviven. Se han asignado más de 100 000 ASN públicos entre ambos tipos. Además existe un rango de ASN privados que no se anuncian a Internet, y que se utilizan por las organizaciones para sus redes internas.

Los AS se interconectan entre sí por medio de un tipo especial de router que se denomina *pasarela de borde* (*border gateway*) y que es capaz de encaminar paquetes a través de varios AS. Internet es una colección de AS interconectados.

La existencia de los AS da lugar a dos tipos diferentes de encaminamiento dinámico:

- **Encaminamiento interno** designa los algoritmos y protocolos de pasarela interior o IGP (Interior Gateway Protocol). Se refiere al encaminamiento de paquetes dentro de un mismo AS. Como cada AS es una unidad independiente, cada administrador puede decidir el algoritmo y protocolo de encaminamiento IGP que más le convenga.
- **Encaminamiento externo** designa los algoritmos y protocolos de pasarela exterior o EGP (Exterior Gateway Protocol). Se refiere al encaminamiento de paquetes entre AS independientes. A diferencia del interno, en este caso, todos los sistemas autónomos deben utilizar el mismo protocolo de encaminamiento.

A su vez, existen dos tipos principales de algoritmos de encaminamiento interno: *vector distancia* y *estado de enlace*. En el caso del encaminamiento exterior se utiliza esencialmente el algoritmo *vector ruta*. Dedicaremos las siguientes secciones a estudiarlos en detalle.

9.3. Topología de referencia: triple delta

Para explicar los algoritmos de encaminamiento interno, utilizaremos una topología de red concreta a modo de ejemplo que llamaremos «triple delta», porque visualmente forma tres triángulos. Esto ayudará a entender sus diferencias y comparar ventajas y desventajas. La topología en cuestión la puedes ver en la Figura 9.1.

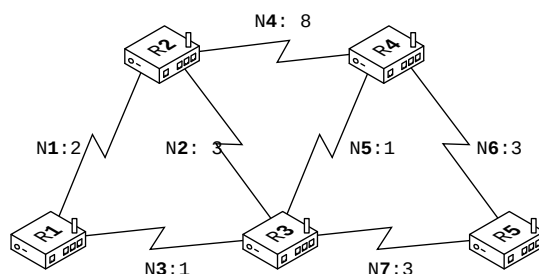


FIGURA 9.1: Topología «triple delta»

Aunque esta topología incluye detalles como la tecnología y tipo de enlaces (serie, en este caso), eso no es importante para el algoritmo de encaminamiento. Lo relevante es el grafo equivalente. En este, los nodos representan a los routers y las aristas a los enlaces. Esto se puede aplicar a cualquier red. Las redes a las que los routers dan acceso tampoco son relevantes para el cálculo de rutas dado que lo importante es poder llegar hasta el router que proporciona acceso a cada red. Por la misma razón, para el algoritmo tampoco son importantes las direcciones de los dispositivos. Verás que cuando se estudian los algoritmos desde un enfoque teórico la columna «destino» de las tablas de encaminamiento no contiene direcciones de red como habitualmente, sino simplemente los nombres de los routers.

Con todo eso, el grafo equivalente a la topología de la Figura 9.1 sería el que se muestra en la Figura 9.2. Como en el original, los enlaces del grafo indican su coste. En este ejemplo, se utiliza una métrica de latencia, pero podría ser cualquier otra y a efectos del algoritmo no es relevante, siempre que el objetivo sea minimizar el coste total de la ruta.

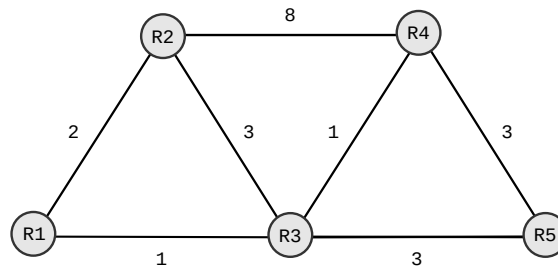


FIGURA 9.2: Grafo equivalente de la topología triple delta

9.4. Redundancia

Esta topología de referencia es intencionadamente **redundante**, es decir, se proporcionan varias caminos para llegar a cualquier destino desde cualquier origen. Las redes e interredes suelen diseñarse de este modo porque aporta una gran ventaja que aprovecha automáticamente la conmutación de paquetes, que es disponer de caminos alternativos. La redundancia es la base para proporcionar tolerancia a fallos y balanceo de carga.

La tolerancia a fallos es la capacidad de la red¹ para seguir funcionando —llevando los paquetes a sus destinos— aún en presencia de fallos. En muchas situaciones, las prestaciones se verán afectas, porque quizá el camino óptimo no esté disponible a causa del fallo, pero esencialmente la red seguirá haciendo su trabajo. Por el modo en que funciona la conmutación de paquetes es muy probable que el servicio no llegue a interrumpirse en ningún momento y los usuarios no noten nada en absoluto. Del mismo modo, si el problema es temporal, la red volverá a funcionar con normalidad cuando se resuelva. En estas situaciones se dice que el servicio proporciona «transparencia de fallos».

El balanceo de carga consiste en aprovechar la redundancia para distribuir el tráfico entre los caminos disponibles. Se trata de utilizar todos los recursos disponibles de forma equitativa evitando la saturación de unos recursos mientras otros quedan infrautilizados. El balanceo de carga puede ser complejo y puede requerir algoritmos capaces de manejar múltiples variables, criterios y estimaciones.

Los algoritmos de encaminamiento dinámico pueden sacar partido de la redundancia y proporcionar tolerancia a fallos y, los más avanzados, también balanceo de carga.

¹Se puede aplicar a cualquier sistema

9.5. La ruta más corta

Todos los algoritmos de encaminamiento tratan de encontrar la mejor ruta (la óptima) desde un nodo origen a un nodo destino. Pero la «mejor ruta» o incluso aunque hablemos de la «ruta más corta» no es necesariamente la ruta que recorre la menor distancia. La «mejor ruta» es la que tiene un menor coste. El coste de la ruta normalmente se calcula como suma del coste de todos los enlaces que la forman. El coste del enlace puede ser cualquier propiedad medible que el administrador considere relevante. Esta propiedad determina la ‘métrica’ del algoritmo. La métrica más sencilla es el número de saltos, pero hay muchas otras: retardo, ancho de banda, fiabilidad, coste económico, etc., o combinación de ellas.

Algunas de estas métricas no cambian con el tiempo. Una ruta de 5 saltos seguirá teniendo 5 saltos independientemente de que la red esté libre o saturada, es decir, no depende de la carga. Otras, como el retardo o el ancho de banda sí dependen; incluso el propio algoritmo de encaminamiento podría influir en esas propiedades. Por ejemplo, si existe un enlace con bajo retardo y el algoritmo la prioriza, es probable que la carga adicional eleve el retardo haciendo que el propio algoritmo la considere menos adecuada en el futuro inmediato. Este es un efecto muy perjudicial porque afecta a la convergencia del algoritmo. Lo deseable es que el algoritmo no modifique las rutas a menos que ocurran cambios en la topología.

El cálculo de la ruta óptima se base en una idea sencilla: el «principio de optimalidad». Este principio dice que si existe una ruta óptima entre el nodo A y el B, que pasa por C, entonces la ruta entre C y B también es necesariamente óptima. En esta idea se fundamentan los algoritmos de Dijkstra, Bellman-Ford y otros, que son los más utilizados en encaminamiento dinámico.

El algoritmo Dijkstra [18] calcula el camino más corto (de menor coste) entre dos nodos de un grafo no dirigido y ponderado, es decir, un grafo cuyas aristas son transitables en ambas direcciones y tienen costes asociados a una métrica arbitraria. La aplicación general del algoritmo al grafo completo calcula la ruta más corta a todos los nodos. Como esas rutas tienen partes comunes, el resultado es el árbol de rutas más cortas (SPT) para el origen solicitado. Por ejemplo, para la topología de referencia anterior, el árbol de rutas más cortas desde R1 sería el de Figura 9.3.

De manera resumida, el algoritmo de Dijkstra hace lo siguiente:

1. Inicializa el coste a todos los nodos a infinito, excepto el nodo origen, que se fija en 0.
2. Marca todos los nodos como no visitados.

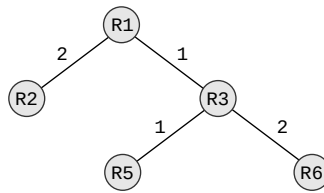


FIGURA 9.3: Árbol de rutas más cortas (SPT) para R1

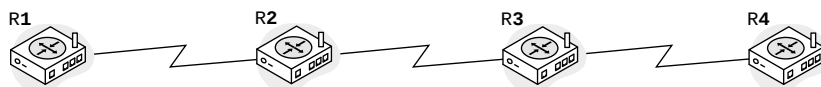
3. Selecciona el nodo no visitado con el menor coste y lo marca como visitado.
4. Actualiza los costes de los nodos vecinos del nodo visitado. Si el nuevo coste es menor que el coste actual, lo actualiza.
5. Repite los pasos 3 y 4 hasta que todos los nodos hayan sido visitados o no queden nodos accesibles.

En el repositorio de ejemplos puedes encontrar una implementación del algoritmo Dijkstra ([🔗/dijkstra/dijkstra.py](#)) y un notebook que lo aplica a un ejemplo ([🔗/dijkstra/dijkstra.ipynb](#)).

9.6. Vector-distancia

Con el algoritmo vector-distancia inicialmente cada router solo tiene información sobre sus vecinos (routers directamente conectados). Esa información, que consiste esencialmente en la dirección/identificador y la distancia/coste a cada vecino, se coloca en una tabla llamada *vector-distancia*. La obtención de esta información es lo que llamamos *inicialización*. Después cada router envía su vector solo a los vecinos, que la utilizan para actualizar sus propios vectores-distancia y sus tablas de encaminamiento. A esa transferencia de vectores es a lo que llamamos una *iteración* del algoritmo. La inicialización no es una iteración puesto que no hay propagación de vectores-distancia.

Veamos un ejemplo trivial con 4 routers empleando una métrica de saltos:



La tabla 9.1 muestra cómo se propagan los vectores-distancia. La primera fila es la mencionada *inicialización*. Los elementos de estos vectores tienen el formato *destino:coste:vecino*, que corresponden respectivamente al router destino, el coste hasta él y el vecino siguiente salto —siendo - la forma de

indicar entrega directa. Por ejemplo, para R1 aparece R1:0:- que significa que el coste para llegar a sí mismo es 0 (lógicamente) y que puede acceder con entrega directa. También aparece R2:1:-, que indica que puede llegar a R2 con un coste de 1 y entrega directa. Fíjate que los vectores incluyen a los propios routers (los que tienen coste 0) dado que esta información debe enviarse también.

	R1	R2	R3	R4
inicialización	R1:0:- R2:1:-	R1:1:- R2:0:- R3:1:-	R2:1:- R3:0:- R4:1:-	R3:1:- R4:0:-
primera iteración	R3:2:R2	R4:2:R3	R1:2:R2	R2:2:R3
segunda iteración	R4:3:R2	-	-	R1:3:R3

CUADRO 9.1: Propagación de vectores-distancia

La segunda y tercera filas muestran la nueva información que incorpora cada router a su vector-distancia en cada iteración del algoritmo. Por ejemplo, en la primera iteración R1 recibe información de R3 a través de R2 y actualiza su vector-distancia añadiendo R3:2:R2, es decir, puede llegar a R3 con un coste de 2 a través de R2. En la tercera R1, después de recibir información procedente de R4 añade R4:3:R2, es decir, puede llegar a R4 con un coste de 3 a través de R2. Por tanto, en esta segunda iteración, los routers de los extremos ya tienen la información necesaria para llegar a todos los demás routers con al menor coste posible. Cuando ocurre esto se dice que el algoritmo ha *convergiado*. En este caso, la convergencia se produce en 2 iteraciones.

Ahora veamos un ejemplo más complejo sobre la topología de referencia de la Figura 9.1 y empleando una métrica basada en los costes de los enlaces. La tabla 9.2 muestra el vector-distancia inicial de cada router (un router por fila).

R1	R1:0:-	R2:2:-	R3:1:-			
R2	R1:2:-	R2:0:-	R3:3:-		R5:8:-	
R3	R1:1:-	R2:3:-	R3:0:-	R4:2:-		
R4			R3:2:-	R4:0:-	R5:1:-	R6:3:-
R5		R2:8:-		R4:1:-	R5:0:-	R6:3:-
R6				R4:3:-	R5:3:-	R6:0:-

CUADRO 9.2: Vector-distancia inicial de cada router

Lo mostramos aquí como una tabla para facilitar su lectura, pero en la práctica son vectores individuales. Es decir, lo que enviaría R1 en la primera iteración sería algo como (R1:0:-), (R2:2:-), (R3:1:-).

Dado que en cada iteración la información solo avanza un salto, si emplea una métrica de saltos, el algoritmo converge en un número de iteraciones igual al diámetro del grafo menos uno. Pero si la métrica es otra, podría requerir más iteraciones, porque rutas con más saltos podrían tener costes menores. El diámetro del grafo indica la cantidad de aristas (saltos) de la más costosa de las rutas menos costosas. Se define formalmente como:

$$D(G) = \max_{u,v \in V} d(u,v) \quad (9.1)$$

donde V es el conjunto de todos los vértices del grafo G y $d(u,v)$ representa el coste del camino óptimo entre los nodos u y v .

La tabla 9.3 muestra los vectores de todos los routers después de la primera iteración. Fíjate que R1 utiliza R2 para llegar a R5 con un coste de 10. Claramente no es la mejor ruta. La ruta óptima es R1-R3-R4-R5, pero en la primera iteración R1 aún no ha recibido el vector de R4. Aún no sabe que existe una ruta mejor para llegar a R5. Puedes comprobar que esta misma situación ocurre con otras rutas.

R1	R1:0:-	R2:2:-	R3:1:-	R4:3:R3	R5:10:R2	
R2	R1:2:-	R2:0:-	R3:3:-	R4:5:R3	R5:8:-	R6:11:R5
R3	R1:1:-	R2:3:-	R3:0:-	R4:2:-	R5:3:R4	R6:5:R4
R4	R1:3:R3	R2:5:R3	R3:2:-	R4:0:-	R5:1:-	R6:3:-
R5	R1:10:R2	R2:8:-	R3:3:R4	R4:1:-	R5:0:-	R6:3:-
R6		R2:11:R5	R3:5:R4	R4:3:-	R5:3:-	R6:0:-

CUADRO 9.3: Vector-distancia de cada router después de la primera iteración

La tabla 9.4 muestra los vectores de cada router después de la segunda iteración. Las celdas sombreadas corresponden a vectores que han mejorado su coste respecto a la primera iteración. En esta situación el algoritmo ha convergido y los vectores no cambiarán a menos que se produzca algún cambio en la topología.

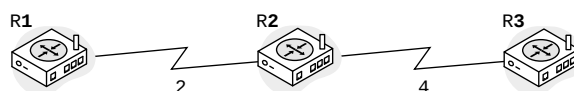
Para esta topología, también han sido necesarias 2 iteraciones (3 saltos entre los routers más distantes). Sin embargo, el algoritmo no acaba al alcanzar la convergencia. Sigue trabajando (ejecutando nuevas iteraciones) indefinidamente porque la topología puede cambiar en cualquier momento. El algoritmo debe adaptar las tablas de encaminamiento de los routers a las condiciones dinámicas de la red.

R1	R1:0:-	R2:2:-	R3:1:-	R4:3:R3	R5:4:R3	R6:6:R3
R2	R1:2:-	R2:0:-	R3:3:-	R4:5:R3	R5:6:R3	R6:8:R3
R3	R1:1:-	R2:3:-	R3:0:-	R4:2:-	R5:3:R4	R6:5:R4
R4	R1:3:R3	R2:5:R3	R3:2:-	R4:0:-	R5:1:-	R6:3:-
R5	R1:4:R4	R2:6:R4	R3:3:R4	R4:1:-	R5:0:-	R6:3:-
R6	R1:6:R4	R2:8:R4	R3:5:R4	R4:3:-	R5:3:-	R6:0:-

CUADRO 9.4: Vector-distancia de cada router tras alcanzar la convergencia

9.6.1. Problemas y limitaciones de vector-distancia

El principal problema del algoritmo de vector-distancia ya lo hemos visto: es lento. Tarda mucho en converger aunque todo vaya bien. Pero cuando las cosas van mal, es mucho más lento. Partamos de una topología muy simple, formada solo por tres routers R1, R2 y R3 y sus vectores-distancia una vez que algoritmo ha convergido.



R1	R1:0:-	R2:2:-	R3:6:R2
R2	R1:2:-	R2:0:-	R3:4:-
R3	R1:6:R2	R2:4:-	R3:0:-

Supongamos ahora que el R3 desaparece o el enlace que lo conecta con R2 se rompe. A partir de ese momento, R1 y R2 intentan utilizar la información disponible para determinar la nueva ruta y coste para llegar a R3. Eso provoca las siguientes acciones:

- R2 ya no recibe información de R3, solo de R1, que le indica que puede llegar a R3 con un coste de 6 (R3:6:R2). R2 actualiza su vector con R3:8:R1, es decir, puede llegar a R3 con un coste de 8 (6+2) a través de R1. Obviamente esto no es cierto, pero R2 no tiene forma de saberlo.
- R1 recibe el nuevo vector de R2, de modo que actualiza su vector a R3:10:R2.
- R2 actualiza su vector a R3:12:R1.
- ...

Este proceso continua indefinidamente, aumentando el coste en cada iteración, y por eso se conoce como *cuenta a infinito* (*count to infinity*). Es un problema grave porque R1 y R2 se estarán reenviando entre sí los paquetes dirigidos a R3 hasta que el valor de TTL llegue a 0 y acaben por descartarse.

Esto se conoce como un *bucle de encaminamiento*, aunque también puede ser provocado por otras causas.

Existen algunas soluciones para el problema de la cuenta a infinito:

- **«Definir» o acotar el infinito.** Consiste en fijar un valor máximo de coste. Un valor mayor implica considerar inaccesible el router afectado. El inconveniente es que limita también el tamaño de la red. Ninguna ruta real podrá tener un coste mayor al valor fijado.
- **Horizonte dividido (*split horizon*).** Consiste en no enviar a un vecino la información obtenida de ese vecino. Para el ejemplo implicaría que R1 no envíe a R2 el vector R3:6:R2, ya que R1 ha aprendido sobre R3 por medio de R2.
- **Ruta inversa envenenada (*poison reverse*).** Se utiliza en combinación con horizonte dividido. Consiste en informar al router del que ha aprendido un destino, que ese destino es inalcanzable para él. En el ejemplo, R1 enviaría a R2 el vector R3: -1:, en el entendido de que -1 indica inalcanzable.

Aunque *horizonte dividido* y *ruta inversa envenenada* mejoran el desempeño del algoritmo, no resuelven completamente el problema de la cuenta a infinito. Esto es porque estas técnicas solo afectan a vecinos, pero no evitan que la información errónea llegue a través de otros routers cuando la topología no es tan simple como la del ejemplo.

Además de la cuenta a infinito, el algoritmo de vector-distancia tiene otros problemas:

- La ya citada convergencia lenta. La información de cada router se propaga solo de uno en uno.
- Los routers envían información de encaminamiento periódicamente aunque no haya cambios en la topología, lo que consume ancho de banda y recursos.
- Puede ser inestable en redes grandes o topologías complejas, provocando cambios continuos en las tablas de encaminamiento.

9.6.2. RIP

Routing Information Protocol (RIP) es un protocolo de encaminamiento que emplea el algoritmo vector-distancia. Fue el protocolo de encaminamiento más utilizado en Internet hasta mediados de los 90. A partir de entonces empezó a ser reemplazado por OSPF, BGP e IS-IS. La primera versión de RIP [19] empleaba el algoritmo de Bellman-Ford para el cálculo de la ruta más corta.

Utiliza una métrica de saltos y para mitigar el problema de la cuenta a infinito define un valor máximo de 15 saltos y aplica horizonte dividido. Es un protocolo *classfull*, es decir, no envía información sobre la máscara de subred y por tanto solo puede trabajar con las redes de clase A, B y C originales.

Envía actualizaciones de encaminamiento (vectores-distancia) cada 30 segundos a la dirección broadcast 255.255.255.255.

El mensaje RIP indica el comando (1 para petición, 2 para respuesta) y la versión del protocolo. Después incluye un máximo de 25 vectores-distancia. Cada uno de ellos consta de:

- Familia de direcciones.
- Dirección IP de la red destino.
- Número de saltos hasta la red destino.

En 1993 se publicó RIP versión 2 [20] que utiliza el algoritmo de Ford-Fulkerson para el cálculo de la ruta óptima. Incluye varias mejoras que solventan los problemas del RIP original. Incluye soporte para VLSM y CIDR, utiliza multicast para enviar las actualizaciones (grupo 224.0.0.9) e incluye un mecanismo de autenticación.

Aunque a día de hoy muchos equipos aún ofrecen soporte, su uso es residual y se limita a redes pequeñas y simples, o entornos de laboratorio.

9.7. Estado de enlace

El «otro» algoritmo importante para encaminamiento interno es estado de enlace (*link state*). La diferencia principal con vector-distancia es que aquí los routers tienen información topológica de toda la subred, y no solo una medida de coste a cada router.

Por su cuenta cada router:

- Descubre sus vecinos y sus direcciones.
- Mide el coste a cada vecino.
- Construye un mensaje con la información obtenida.
- Envía ese mensaje a todos los routers de la subred (o AS).
- Con la información propia y la recibida crea el grafo de la subred.
- Calcula la ruta más corta a todos los routers.

Cuando un enlace o router cambia de estado, es decir, se activa, desactiva o cambia su coste, los routers que lo detecten envían inmediatamente una actualización a todos los demás.

Este algoritmo requiere más cantidad de memoria al tener que almacenar la topología, requiere más computación al tener que calcular la ruta a cada destino y más ancho de banda ya que genera más tráfico. A cambio, converge más rápido, es más estable y se puede utilizar en subredes más grandes y complejas que vector-distancia.

9.7.1. OSPF

Open Short Path First (OSPF) es el protocolo de encaminamiento de estado de enlace más utilizado en Internet en la actualidad. La primera versión funcional (la 2) fue publicada en 1991 [21] y se diseñó para sustituir a RIP, resolviendo la mayoría de sus problemas, aunque a costa de mayor complejidad y consumo de recursos. Es un protocolo *classless* desde su diseño y por tanto soporta CIDR y VLSM. Utiliza Dijkstra para el cálculo de la ruta más corta y ancho de banda como métrica.

Segmenta el SA en áreas para mejorar la escalabilidad y reducir el tráfico del propio protocolo. La distribución de áreas es jerárquica y hay dos niveles: áreas de tránsito y áreas de borde. El área *backbone* (la 0) es la principal área de tránsito y debe estar en cualquier despliegue de OSPF. Las áreas de tránsito conectan con las de borde routers ABR (Area Border Router), es decir, el tráfico de las otras áreas debe pasar por el área *backbone*. Las áreas de borde solo manejan tráfico propio. Utilizan los ABR para comunicarse con el resto del AS y los ASBR (Autonomous System Border Router) para comunicarse con otros AS.

Cada router mantiene una LSDB (Link State Database) con la información de la topología. Se actualiza periódicamente o cuando se producen cambios a partir del intercambio de mensajes LSA. Hay 7 tipos distintos de LSA en función del tipo de router que lo envía, la información que contiene y su alcance. Los mensajes LSA se envían al grupo multicast 224.0.0.5.

9.8. Vector-ruta y BGP

A diferencia de vector-distancia y estado de enlace, vector-ruta es un algoritmo de encaminamiento externo, es decir, se utiliza para encaminamiento entre AS. Los EGP son los encargados de este tipo de encaminamiento. BGP (Border Gateway Protocol) es el EGP más utilizado. Obviamente hay muchas diferencias entre los EGP, que utilizan vector-ruta, respecto a los IGP, pero hay dos que resultan muy evidentes:

- Las rutas se determinan entre AS, no entre routers.
- Las tablas indican la ruta completa hasta el AS destino, no solo el siguiente salto. Estas sí son literalmente «tablas de rutas».

Como vimos en la §9.2, cada AS se identifica con un ASN. Una ruta a través de distintos AS se especifica con una secuencia de ASN (se llama *ASPATH*). Los routers de frontera (*border router*), como su nombre indica, son los que conectan un AS a otros. Son los encargados de anunciar las *ASPATH* a los demás AS. Cuando un router BGP recibe una ruta de este tipo puede comprobar fácilmente si esta pasa por su AS. Si ese es el caso, la descarta, y de ese modo evita bucles de encaminamiento. En caso contrario, añade su ASN al final y anuncia esa nueva ruta a los demás AS. La métrica más sencilla para medir la bondad de una ruta es su longitud. Una ruta será mejor si implica menos AS. Las preferencias del administrador son determinantes en la elección, mucho más que en los IGP, incluso por encima de las métricas puramente técnicas.

Fíjate que los routers de frontera tienen que ejecutar tanto un IGP como un EGP. Un IGP (por ejemplo OSPF para encaminar los paquetes dentro del AS) y un EGP (como BGP) para encaminarlos entre AS. Además BGP está compuesto por dos protocolos: *eBGP* es el que comunica rutas BGP entre distintos AS, mientras que *iBGP* lo hace entre los routers de un mismo AS.

En la Figura 9.4 aparecen tres AS interconectados. Los marcados en amarillo son routers frontera que tiene que ejecutar tanto un IGP como un EGP.

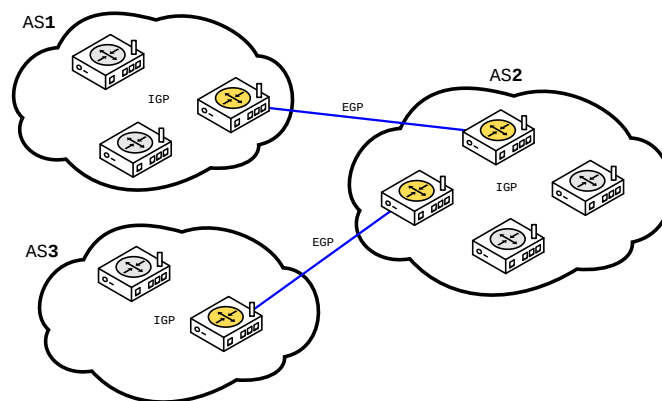


FIGURA 9.4: Sistemas Autónomos interconectados

9.9. Laboratorio de encaminamiento Delta

Para ver los protocolos de encaminamiento desde un punto de vista mucho más práctico, vamos a montar un pequeño laboratorio emulando la topología de la Figura 9.5, que llamaremos «Delta».

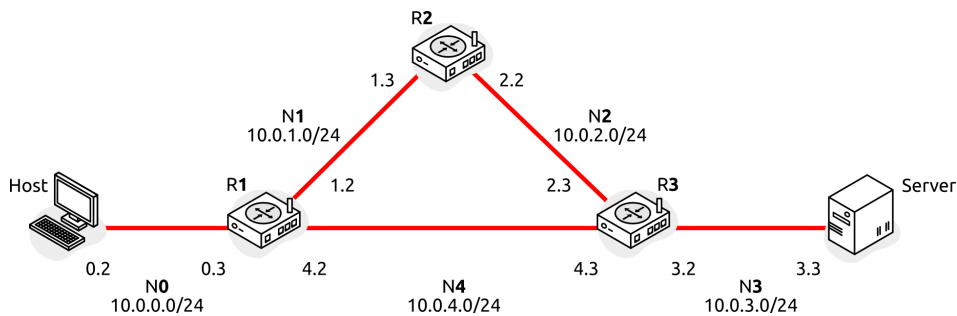


FIGURA 9.5: Topología Delta

Aunque es una topología ciertamente muy simple, tiene una característica importante: es redundante. Algo que, como ya hemos visto, es siempre buena idea. Por eso los protocolos de encaminamiento deben encontrar la mejor de las rutas disponibles. Sin embargo, los protocolos actuales no descartan las rutas no elegidas. Es mejor idea mantenerlas como rutas alternativas para poder utilizarlas en caso de fallo o para balancear el tráfico.

El nodo Host de la figura es tu PC real. Los routers y el nodo Server son contenedores docker. Eso significa que te puedes comunicar con ellos ejecutando comandos reales en tu terminal.

Todos los archivos necesarios para ejecutar este laboratorio (llamado `lab-routing-delta`) están disponibles en el repositorio de ejemplos del libro (página XXIX).

La topología completa está especificada en el archivo `compose.yml` (Listado 9.1).

```
x-common: &common-settings
  privileged: true
  restart: "no"
  ulimits:
    nofile:
      soft: 100000
      hard: 200000

services:
  r1:
    image: lab-delta-router
    container_name: r1
    hostname: r1
```

```
<<: *common-settings
volumes:
  - ./daemons:/etc/frr/daemons
  - ./frr-r1.conf:/etc/frr/frr.conf
networks:
  N0: { ipv4_address: 10.0.0.3 }
  N1: { ipv4_address: 10.0.1.2 }
  N4: { ipv4_address: 10.0.4.2 }

r2:
  image: lab-delta-router
  container_name: r2
  hostname: r2
  <<: *common-settings
  volumes:
    - ./daemons:/etc/frr/daemons
    - ./frr-r2.conf:/etc/frr/frr.conf
  networks:
    N1: { ipv4_address: 10.0.1.3 }
    N2: { ipv4_address: 10.0.2.2 }

r3:
  image: lab-delta-router
  container_name: r3
  hostname: r3
  <<: *common-settings
  volumes:
    - ./daemons:/etc/frr/daemons
    - ./frr-r3.conf:/etc/frr/frr.conf
  networks:
    N2: { ipv4_address: 10.0.2.3 }
    N3: { ipv4_address: 10.0.3.2 }
    N4: { ipv4_address: 10.0.4.3 }

server:
  image: lab-delta-server
  container_name: server
  privileged: true
  networks:
    N3: { ipv4_address: 10.0.3.3 }

networks:
  N0:
    driver: bridge
    driver_opts: { com.docker.network.bridge.name: N0 }
    ipam: { config: [{subnet: 10.0.0.0/24}] }
  N1:
    driver: bridge
    driver_opts: { com.docker.network.bridge.name: N1 }
    ipam: { config: [{subnet: 10.0.1.0/24}] }
  N2:
    driver: bridge
    driver_opts: { com.docker.network.bridge.name: N2 }
    ipam: { config: [{subnet: 10.0.2.0/24}] }
  N3:
    driver: bridge
    driver_opts: { com.docker.network.bridge.name: N3 }
    ipam: { config: [{subnet: 10.0.3.0/24}] }
  N4:
```

```
driver: bridge
driver_opts: { com.docker.network.bridge.name: N4 }
ipam: { config: [{subnet: 10.0.4.0/24}] }
```

LISTADO 9.1: Descripción del laboratorio de encaminamiento Delta

 lab-routing-delta/compose.yml

Este archivo se procesa con el comando `docker-compose`². Además se necesitan otros archivos de configuración y scripts.

Se proporciona un archivo `Makefile` en el mismo directorio que ejecuta algunas tareas básicas necesarias —que se explican a continuación. Es importante utilizarlo en lugar de lanzar directamente los comandos de `docker`. Necesitas instalar el paquete `Debian make` si no lo tienes. Para arrancar la **configuración básica** del laboratorio ejecuta:

```
$ make up
[...]
[+] Running 4/4
   Container r1      Running
   Container r2      Running
   Container r3      Running
   Container server   Running
```

Una vez todo en marcha, puedes ver los contenedores con:

```
$ docker compose ps
```

NAME	IMAGE	COMMAND	SERVICE	CREATED	STATUS
r1	router	"sh -c 'ip route del..."	r1	26 seconds ago	Up
r2	router	"sh -c 'ip route del..."	r2	26 seconds ago	Up
r3	router	"sh -c 'ip route del..."	r3	26 seconds ago	Up
server	server	"sh -c 'ip route del..."	server	26 seconds ago	Up

Y las redes con:

```
$ docker network ls
```

NETWORK ID	NAME	DRIVER	SCOPE
266d2cb29af4	bridge	bridge	local
059b37dd8ae6	host	host	local
030f578f0ff0	none	null	local
cbcf75bf8048	lab-routing-delta_N0	bridge	local
ab4139a63510	lab-routing-delta_N1	bridge	local
2520be21d05b	lab-routing-delta_N2	bridge	local
6fe6bdc69c3f	lab-routing-delta_N3	bridge	local
270e2a372326	lab-routing-delta_N4	bridge	local

Estas redes se implementan como *bridges*. Un bridge es una especie de conmutador (*switch*) virtual que proporciona Linux y que funciona como una red Ethernet. Docker crea un *bridge* por cada red (de N0 a N4). También crea interfaces Ethernet virtuales para conectar los contenedores a las redes. Todo esto lo puedes ver si ejecutas el comando `ip` a en tu PC. Aquí se

²En versiones más recientes puede estar disponible como `docker compose` (sin guion).

muestra un fragmento de lo que verás. Los `br-<número>` son los *bridge* y los `veth<número>` son las interfaces virtuales.

```

1 $ ip a
2 767: br-203457108c6e: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 state UP group default
3     link/ether d2:aa:22:09:37:28 brd ff:ff:ff:ff:ff:ff
4     inet 10.0.0.1/24 brd 10.0.0.255 scope global br-203457108c6e
5         valid_lft forever preferred_lft forever
6 768: veth28e8f66@if2: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 master br-203457108c6e
7     ↪ state UP group default
        link/ether fe:87:85:50:cb:f8 brd ff:ff:ff:ff:ff:ff link-netnsid 0

```

Vale la pena mencionar lo que hace `make up` del `Makefile` para evitar confusiones. Es esta:

```

up: build
    docker compose up --remove-orphans -d
    sudo ./rm-bridge-addr.sh
    sudo ./setup-hosts.sh

```

Veamos su función:

- `docker compose up` arranca los contenedores y los mantiene en *background*. También crea las redes que acabamos de comentar.
- `rm-bridge-addr.sh` elimina las direcciones IP de esos bridge, excepto el que conecta el host con la red `N0`. Esto es necesario porque de otro modo tu PC tendría acceso directo a todas las redes invalidando el ejercicio. Lo que nosotros pretendemos es que sean los routers los que lleven el tráfico a esas redes.
- `setup-hosts.sh` configura en tu PC la ruta estática que envía a `r1` el tráfico dirigido a `10.0.0.0/16` (todas las redes de la figura). También configura `r3` como router por defecto para `Server`.

Ahora puedes ejecutar comandos en cualquiera de los contenedores. Por ejemplo, puedes ver la tabla de encaminamiento de `r1` con el siguiente comando:

```

$ docker exec r1 ip route
10.0.0.0/24 dev eth2 proto kernel scope link src 10.0.0.3
10.0.1.0/24 dev eth0 proto kernel scope link src 10.0.1.2
10.0.4.0/24 dev eth1 proto kernel scope link src 10.0.4.2

```

Las filas de esta tabla las ha creado automáticamente el SO al activar las interfaces del router. Estamos seguros de eso porque la propia tabla lo indica con *proto kernel*. Ambas filas indican que para llegar a las redes `N0`, `N1` y `N4` se debe hacer entrega directa a través de las interfaces `eth2`, `eth0` y `eth1` respectivamente. Docker asigna los nombres a estas interfaces automáticamente, y podrían cambiar en cada ejecución, así que quizá no

coincidan con los que ves en tu PC. Puedes probar el mismo comando con r2 y r3 para ver sus tablas.

A diferencia de los routers, el nodo **Server** solo tiene una interfaz y está conectada a la red **N3**. Mira el resultado en este caso:

```
$ docker exec server ip route
10.0.3.0/24 dev eth0 proto kernel scope link src 10.0.3.3
default via 10.0.3.2 dev eth0
```

Esta tabla de encaminamiento solo tiene una fila para entrega directa a la red **N3**. La segunda fila la ha configurado el script `setup-hosts.sh`, del que hablamos antes, indicando que r3 (10.0.3.2) es su router por defecto. Solo de ese modo podrá enviar tráfico a las otras redes.

Con esta configuración básica, el nodo **Server** tiene conectividad con r3:

```
$ docker exec server ping -c1 10.0.3.2
PING 10.0.3.2 (10.0.3.2) 56(84) bytes of data.
64 bytes from 10.0.3.2: icmp_seq=1 ttl=64 time=0.188 ms
```

Pero no con las otras redes, porque los routers no las conocen:

```
$ docker exec server ping -c1 10.0.1.2
PING 10.0.1.2 (10.0.1.2) 56(84) bytes of data.
From 10.0.3.2 icmp_seq=1 Destination Net Unreachable
```

En las siguientes secciones vamos a configurar el encaminamiento de esta interred de tres formas: encaminamiento estático, RIP y OSPF.

9.9.1. Encaminamiento estático

El encaminamiento estático es muy sencillo en este caso. Simplemente hay que decirle a los routers cómo llegar a las redes distantes, es decir, a las que no están directamente conectados. Por ejemplo, a r1 debes indicarle cómo llegar a las redes **N2** y **N3**. Los comandos necesarios se incluyen en el script `setup-static.sh`.

```
1 docker exec r1 ip route add default via 10.0.4.3
2 docker exec r2 ip route add 10.0.0.0/24 via 10.0.1.2
3 docker exec r2 ip route add default via 10.0.2.3
4 docker exec r3 ip route add default via 10.0.4.2
```

Fíjate que r3 se configura como router por defecto para r1, porque él sabrá cómo llegar a las otras 2 redes (**línea 1**). Hacemos algo equivalente también con r3 para r2 (**línea 3**) y con r1 para r3 (**línea 4**). Sin embargo, para que r2 pueda llegar a la red **N0** debemos indicar explícitamente que lo envíe a través de r1 (**línea 2**), porque no puede tener dos routers por defecto.

Para reiniciar la topología, aplicar los scripts que hemos visto y configurar el encaminamiento estático, ejecuta:

```
$ make static
```

Ahora puedes comprobar que tienes conectividad con **Server**. Desde una consola en tu PC puedes probarle ping:

```
$ ping -c1 10.0.3.3
PING 10.0.3.3 (10.0.3.3) 56(84) bytes of data.
64 bytes from 10.0.3.3: icmp_seq=1 ttl=64 time=0.161 ms
```

O **traceroute**:

```
$ traceroute 10.0.3.3
traceroute to 10.0.3.3 (10.0.3.3), 30 hops max, 60 byte packets
 1  10.0.0.3 (10.0.0.3)  0.767 ms  0.666 ms  0.635 ms
 2  10.0.4.3 (10.0.4.3)  0.608 ms  0.535 ms  0.496 ms
 3  10.0.3.3 (10.0.3.3)  0.459 ms  0.377 ms  0.330 ms
```

En esta salida de **traceroute** se puede ver cómo contestan **r1**, **r3** y el nodo **Server** para cada uno de los 2 saltos necesarios.

También puedes arrancar un servidor en **Server** y acceder a él desde tu PC para comprobar así la conectividad a nivel de transporte. Con el siguiente comando puedes arrancar un servidor **ncat** que simplemente envía la hora al cliente que conecte:

```
$ docker exec server ncat -lp 2000 -c date
```

Y en otra consola, ejecuta el cliente (también con **ncat**):

```
$ ncat 10.0.3.3 2000
Mon Mar  2 20:08:19 CET 2026
```

Con estas tres pruebas no queda ninguna duda de que el encaminamiento está haciendo su trabajo.

9.9.2. Zebra

Actualmente³ muchos de los SO basados en GNU/Linux utilizan un servicio llamado **FRR** (Free Range Routing) para implementar encaminamiento dinámico. **FRR** ofrece un servicio para cada uno de los protocolos de encaminamiento soportados. Todos ellos se comunican con otro servicio llamado **Zebra**, qué es quién coordina y decide qué información se incluye finalmente en la tabla de encaminamiento, permitiendo incluso que varios protocolos trabajen a la vez.

³Escribo esto en 2026.

Los contenedores de los routers ejecutan Zebra, aunque no lo hemos utilizado en la sección anterior. Lo puedes ver en el archivo `router/Dockerfile`:

```
CMD ["sh", "-c", \
    "ip route del default && \
    sysctl -w net.ipv4.ip_forward=1 && \
    /usr/lib/frr/zebra"]
```

Cuando el servicio arranca, elimina la ruta por defecto que configura docker para evitar que interfiera con nuestro esquema de encaminamiento, activa el reenvío y ejecuta zebra, que queda en ejecución mientras el contenedor esté activo. Es posible utilizar `vttysh`⁴ para contactar con Zebra y obtener información o configurar parámetros de encaminamiento. Con el siguiente comando puedes ver la tabla de encaminamiento de `r1` tal como quedó configurada en la sección anterior:

```
$ docker exec -it r1 vtysh
r1# show ip route
Codes: K - kernel route, C - connected, S - static, R - RIP,
       O - OSPF, I - IS-IS, B - BGP, E - EIGRP, N - NHRP,
       T - Table, v - VNC, V - VNC-Direct, A - Babel, F - PBR,
       f - OpenFabric,
       > - selected route, * - FIB route, q - queued, r - rejected, b - backup
       t - trapped, o - offload failure

K>* 0.0.0.0/0 [0/0] via 10.0.4.3, eth2, 00:04:14
C>* 10.0.0.0/24 is directly connected, eth0, 00:04:18
C>* 10.0.1.0/24 is directly connected, eth1, 00:04:18
C>* 10.0.4.0/24 is directly connected, eth2, 00:04:18
```

Puedes ejecutar un comando de `vttysh` directamente con la opción `-c` como en `docker exec r1 vtysh -c "show ip route"`. En la salida verás las 3 redes directamente conectadas (C) y la ruta por defecto (K) que hemos configurado. Al final de cada fila aparece el tiempo transcurrido desde que se estableció.

9.9.3. Encaminamiento RIP

Es momento de configurar RIP en nuestra topología. Partiendo del despliegue original, es necesario proporcionar un archivo de configuración para cada uno de los routers. Por ejemplo, para el router `r1` el archivo es:

```
router rip
 network 10.0.0.0/24
 network 10.0.1.0/24
 network 10.0.4.0/24
```

⁴ctysh está incluido en el paquete `frr`.

LISTADO 9.2: Configuración RIP en el laboratorio de encaminamiento Delta
📄/lab-routing-delta/rip/R1-ripd.conf

Esta configuración le indica al router las redes que debe anunciar. Los 3 routers anuncian las 3 redes a las que están conectados. Para utilizar el protocolo RIP, el router debe ejecutar el servicio `ripd`. El archivo de configuración lo montamos en el propio archivo `compose.yml`. Para ejecutar el servicio `ripd` usamos también `docker exec`. El script `setup-rip.sh` lo hace para los 3 routers.

```
$ docker exec $router /usr/lib/frr/ripd -f /etc/frr/ripd.conf -d
```

Para reiniciar la topología, aplicar los scripts que hemos visto y configurar el encaminamiento RIP simplemente ejecuta:

```
$ make rip
```

Puedes consultar la configuración de cualquiera de los routers con `vttysh`:

```
$ docker exec r3 vtysh -c "show running-config"
Building configuration...

Current configuration:
!
frr version 8.4.4
frr defaults traditional
hostname r3
no ipv6 forwarding
service integrated-vtysh-config
!
router rip
 network 10.0.2.0/24
 network 10.0.3.0/24
 network 10.0.4.0/24
exit
!
end
```

Ahora espera hasta que RIP converja. Para eso puedes simplemente hacer `ping` a `Server` desde tu PC y esperar. Puede tardar alrededor de un minuto. Hasta que las rutas se establezcan, los mensajes no podrán llegar hasta `Server` y verás mensajes de error que te lo indican. Cuando RIP haya establecido las rutas, `ping` empezará a funcionar.

```
$ ping 10.0.3.3
PING 10.0.3.3 (10.0.3.3) 56(84) bytes of data.
From 10.0.0.3 icmp_seq=1 Destination Net Unreachable
From 10.0.0.3 icmp_seq=2 Destination Net Unreachable
From 10.0.0.3 icmp_seq=27 Destination Net Unreachable
64 bytes from 10.0.3.3: icmp_seq=47 ttl=62 time=0.092 ms
64 bytes from 10.0.3.3: icmp_seq=48 ttl=62 time=0.072 ms
64 bytes from 10.0.3.3: icmp_seq=49 ttl=62 time=0.075 ms
```


En ese momento puedes comprobar la tabla de encaminamiento de cualquiera de los routers con `ip route`:

```
$ docker exec r3 ip route
10.0.0.0/24 nhid 9 via 10.0.4.2 dev eth2 proto rip metric 20
10.0.1.0/24 nhid 10 via 10.0.2.2 dev eth0 proto rip metric 20
10.0.2.0/24 dev eth0 proto kernel scope link src 10.0.2.3
10.0.3.0/24 dev eth1 proto kernel scope link src 10.0.3.2
10.0.4.0/24 dev eth2 proto kernel scope link src 10.0.4.3
```

Fíjate en el *proto rip* en las dos primeras filas. Indica que esas rutas las ha aprendido por medio de RIP. También puedes comprobar la tabla de encaminamiento de `r1` con `vttysh`:

```
$ docker exec r3 vtysh -c "show ip route"
Codes: K - kernel route, C - connected, S - static, R - RIP,
       O - OSPF, I - IS-IS, B - BGP, E - EIGRP, N - NHRP,
       T - Table, v - VNC, V - VNC-Direct, A - Babel, F - PBR,
       f - OpenFabric,
       > - selected route, * - FIB route, q - queued, r - rejected, b - backup
       t - trapped, o - offload failure

R>* 10.0.0.0/24 [120/2] via 10.0.4.2, eth2, weight 1, 00:00:53
R>* 10.0.1.0/24 [120/2] via 10.0.2.2, eth0, weight 1, 00:00:53
C>* 10.0.2.0/24 is directly connected, eth0, 00:00:54
C>* 10.0.3.0/24 is directly connected, eth1, 00:00:54
C>* 10.0.4.0/24 is directly connected, eth2, 00:00:54
```

También puedes comprobar el estado del protocolo RIP con `vttysh`:

```
$ docker exec r3 vtysh -c "show ip rip"
Codes: R - RIP, C - connected, S - Static, O - OSPF, B - BGP
Sub-codes:
       (n) - normal, (s) - static, (d) - default, (r) - redistribute,
       (i) - interface

Network        Next Hop        Metric From      Tag Time
R(n) 10.0.0.0/24 10.0.4.2         2 10.0.4.2        0 02:45
R(n) 10.0.1.0/24 10.0.2.2         2 10.0.2.2        0 02:50
C(i) 10.0.2.0/24 0.0.0.0          1 self            0
C(i) 10.0.3.0/24 0.0.0.0          1 self            0
C(i) 10.0.4.0/24 0.0.0.0          1 self            0
```

Prueba también `vttysh -c "show ip rip status"` para obtener información adicional.

Fíjate que aunque hay dos formas de llegar desde `r3` a la red `N1` o desde `r1` a la red `N2`, RIP solo anuncia una de ellas.

Igual que para encaminamiento estático, puedes comprobar la conectividad con `Server` (o cualquiera de las routers) desde tu PC mediante `ping`, `traceroute` o `ncat`.

9.9.4. Encaminamiento OSPF

Por último, vamos a configurar OSPF en la topología Delta, partiendo del despliegue básico. Es necesario proporcionar un archivo de configuración para el servicio `ospfd`, aunque en este caso podemos usar el mismo archivo para los 3 routers. El archivo es `ospfd.conf` y se monta también en el archivo `compose.yml`. El archivo contiene simplemente:

```
router ospf
  redistribute connected
  network 0.0.0.0/0 area 0
\caption[Configuración de \ac{OSPF} en el laboratorio de encaminamiento Delta]
{Configuración \ac{OSPF} en el laboratorio de encaminamiento Delta\}
\codexample{lab-routing-delta/ospfd.conf}}
```

Puedes reiniciar la topología y configurar OSPF con:

```
$ make ospf
```

Como antes, puedes consultar la configuración de cualquiera de los routers con `vttysh`:

```
$ docker exec r3 vtysh -c "show running-config"
Building configuration...

Current configuration:
!
frr version 8.4.4
frr defaults traditional
hostname r3
no ipv6 forwarding
service integrated-vtysh-config
!
router ospf
  redistribute connected
  network 0.0.0.0/0 area 0
exit
!
end
```

Y también, cuando el protocolo haya convergido, puedes comprobar las tablas de encaminamiento con `ip route`. En este caso pueden pasar hasta 2 minutos hasta que las tablas se estabilicen.

```
$ docker exec r3 ip route
10.0.0.0/24 nhid 16 via 10.0.4.2 dev eth2 proto ospf metric 20
10.0.1.0/24 nhid 17 proto ospf metric 20
    nexthop via 10.0.4.2 dev eth2 weight 1
    nexthop via 10.0.2.2 dev eth0 weight 1
10.0.2.0/24 dev eth0 proto kernel scope link src 10.0.2.3
10.0.3.0/24 dev eth1 proto kernel scope link src 10.0.3.2
10.0.4.0/24 dev eth2 proto kernel scope link src 10.0.4.3
```

Aquí puedes ver que OSPF, a diferencia de RIP, proporciona dos formas de llegar a la red N1: a través de r1 y de r2. También puedes verlo con vtysh:

```
$ docker exec r3 vtysh -c "show ip route"
Codes: K - kernel route, C - connected, S - static, R - RIP,
        O - OSPF, I - IS-IS, B - BGP, E - EIGRP, N - NHRP,
        T - Table, v - VNC, V - VNC-Direct, A - Babel, F - PBR,
        f - OpenFabric,
        > - selected route, * - FIB route, q - queued, r - rejected, b - backup
        t - trapped, o - offload failure

O>* 10.0.0.0/24 [110/20] via 10.0.4.2, eth2, weight 1, 00:01:35
O>* 10.0.1.0/24 [110/20] via 10.0.2.2, eth0, weight 1, 00:01:35
    *                via 10.0.4.2, eth2, weight 1, 00:01:35
O  10.0.2.0/24 [110/10] is directly connected, eth0, weight 1, 00:02:20
C>* 10.0.2.0/24 is directly connected, eth0, 00:02:20
O  10.0.3.0/24 [110/10] is directly connected, eth1, weight 1, 00:02:20
C>* 10.0.3.0/24 is directly connected, eth1, 00:02:20
O  10.0.4.0/24 [110/10] is directly connected, eth2, weight 1, 00:02:20
C>* 10.0.4.0/24 is directly connected, eth2, 00:02:20
```

Fíjate que OSPF también anuncia las redes directamente conectadas, pero Zebra prioriza las que configuró el SO (las que llevan el prefijo >).

9.9.5. Reacción ante fallos

Vamos a ver ahora cómo reaccionan los protocolos de encaminamiento ante fallos. Está claro que la ruta óptima para llegar de Host a Server es Host->r1->r2->Server. Ya lo has comprobado con traceroute.

```
$ traceroute 10.0.3.3
traceroute to 10.0.3.3 (10.0.3.3), 30 hops max, 60 byte packets
 1  10.0.0.3 (10.0.0.3)  0.118 ms  0.038 ms  0.031 ms
 2  10.0.4.3 (10.0.4.3)  0.097 ms  0.083 ms  0.053 ms
 3  10.0.3.3 (10.0.3.3)  0.086 ms  0.068 ms  0.065 ms
```

Ahora vamos a desactivar la interfaz de r1 en la red N4 (la que tiene asignada la 10.0.4.2) para ver cómo OSPF reacciona a este «fallo». Veamos primero qué interfaz es la que tiene esa dirección IP:

```
$ docker exec r1 ip a
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 state UNKNOWN group default qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
        valid_lft forever preferred_lft forever
2: eth0@if930: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 state UP group default
    link/ether 5e:a0:fb:3e:84:4b brd ff:ff:ff:ff:ff:ff link-netnsid 0
    inet 10.0.1.2/24 brd 10.0.1.255 scope global eth0
        valid_lft forever preferred_lft forever
3: eth1@if933: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 state UP group default
    link/ether 2a:52:31:cc:59:20 brd ff:ff:ff:ff:ff:ff link-netnsid 0
    inet 10.0.4.2/24 brd 10.0.4.255 scope global eth1
```

```
valid_lft forever preferred_lft forever
4: eth2@if935: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 state UP group default
    link/ether 76:54:4a:3d:54:dc brd ff:ff:ff:ff:ff:ff link-netnsid 0
    inet 10.0.0.3/24 brd 10.0.0.255 scope global eth2
        valid_lft forever preferred_lft forever
```

La interfaz que buscamos es **eth1**. Para desactivarla, ejecuta:

```
$ docker exec r1 ip link set eth1 down
```

Comprueba que está desactivada (*DOWN*) con:

```
$ docker exec r1 ip link show eth1
3: eth1@if520: <BROADCAST,MULTICAST> mtu 1500 state DOWN mode DEFAULT group default
```

Después de converger, comprueba qué ruta siguen los paquetes hasta **Server**:

```
$ traceroute 10.0.3.3
traceroute to 10.0.3.3 (10.0.3.3), 30 hops max, 60 byte packets
 1  10.0.0.3 (10.0.0.3)  0.116 ms  0.040 ms  0.031 ms
 2  10.0.1.3 (10.0.1.3)  0.071 ms  0.050 ms  0.047 ms
 3  10.0.2.3 (10.0.2.3)  0.083 ms  0.065 ms  0.062 ms
 4  10.0.3.3 (10.0.3.3)  0.094 ms  0.079 ms  0.078 ms
```

Ahora la ruta que sigue es **Host->r1->r2->r3->Server**. OSPF ha reaccionado al fallo en **N4** y ha elegido la ruta alternativa. Puedes hacer este mismo ejercicio con **RIP** y verás que también se adapta al fallo.

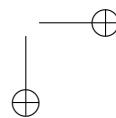
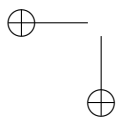
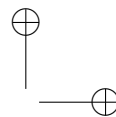
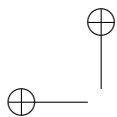
Y ¿qué más?

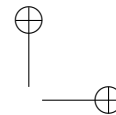
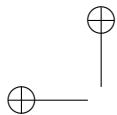
El encaminamiento dinámico permite a las interredes descubrir como es su topología, calcular las rutas óptimas a cada destino y adaptarse a los cambios que se produzcan, sean estos debidos a fallos puntuales o a la aparición de nuevos equipos, enlaces o rutas.

Hemos visto el funcionamiento de los algoritmos más elementales: vector-distancia, estado de enlace y vector-ruta, y hemos estudiado sus problemas y limitaciones. Hemos visto en funcionamiento implementaciones reales de los protocolos **RIP** y **OSPF**, y del sistema **FRR** de **Linux**, gracias a despliegues **docker** sencillos, pero funcionales.

Sin encaminamiento dinámico sería imposible construir interredes de gran tamaño, incluso de mediano tamaño, y obviamente **Internet** no podría existir.

En todo caso, el encaminamiento dinámico genérico (como el que hemos visto aquí) tiene sus limitaciones: se centra en encontrar rutas óptimas y adaptarse a los cambios, pero no contempla las necesidades particulares de usuarios o servicios. Existen muchas aplicaciones: videoconferencia, streaming de video, VoIP, tráfico en tiempo real; que son muy sensibles a la latencia, el jitter o cambios imprevistos en las condiciones del tráfico. Para paliar estas limitaciones existe toda una disciplina: la Calidad de Servicio. Hablaremos de ello en el capítulo 16.





Capítulo 10

Configuración IP

Al terminar este capítulo, entenderás:

- Cuál es la información esencial para que un nodo pueda comunicarse en una red IP.
- Cómo se configura un nodo de forma manual.
- Cómo se configura un nodo de forma automática con DHCP.

Cuando un nodo arranca y el SO accede a las interfaces de red conectadas, puede determinar sus direcciones MAC¹, porque esas direcciones físicas están grabadas en una memoria ROM en el propio periférico.

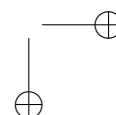
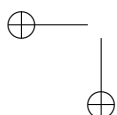
Sin embargo, eso no ocurre con la dirección IP. La dirección IP es parte de la configuración del nodo, y como hemos visto, es totalmente dependiente de la red a la cada NIC esté conectada. Sin una dirección IP válida, el nodo no podrá comunicarse con otros nodos ni en la misma red ni fuera de ella.

10.1. Configuración manual

Si no hay disponible un servicio que ayude a realizar esta configuración automáticamente (que veremos más tarde), el usuario del nodo debe realizar la configuración de forma manual. En ese caso, el administrador de la red tiene que proporcionar al usuario la dirección IP para ese nodo y la máscara de subred. Probar asignando una dirección a ciegas es poco probable que funcione, y puede provocar problemas de conectividad a otros usuarios si se elige una dirección en uso.

Supón que el administrador de la red te ha asignado la dirección **20.3.4.13/24** para la interfaz Ethernet de tu computador. Lo primero que necesitas es saber cómo se llama la interfaz, algo que puedes averiguar con el comando `ip addr` (Figura 10.1).

¹Este es el caso de Ethernet o WiFi, pero no todas las NIC tienen dirección MAC, solo las que utilizan una tecnología de medio compartido.



```
$ sudo ip addr
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue group default qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
        valid_lft forever preferred_lft forever
    inet6 ::1/128 scope host noprefixroute
        valid_lft forever preferred_lft forever
2: eno1: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc fq_codel state UP group default
    qlen 1000
    link/ether f8:5f:2a:c0:ff:ee brd ff:ff:ff:ff:ff:ff
```

FIGURA 10.1: Salida del comando `ip addr` para mostrar las interfaces de red.

Aquí puedes ver la interfaz loopback (`lo`) y la interfaz Ethernet (`link/ether`), que en este caso se llama `eno1`. Ahora puedes asignarle la nueva dirección y ver el resultado:

```
$ sudo ip addr add 20.3.4.13/24 dev eno1
$ sudo ip addr show eno1
2: eno1: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc fq_codel state UP group default
    qlen 1000
    link/ether f8:5f:2a:c0:ff:ee brd ff:ff:ff:ff:ff:ff
    inet 20.3.4.13/24 scope global eno1
        valid_lft forever preferred_lft forever
```

Al hacer esto, el SO crea automáticamente una entrada en la tabla de encaminamiento que indica cómo llegar a los vecinos:

```
$ ip route
20.3.4.0/24 dev eno1 proto kernel scope link src 20.3.4.13
```

En esta situación el nodo tiene conectividad con cualquier vecino de la red local, aunque no puede llegar más allá. Como ya hemos visto, para salir de la red local, se necesita la IP del router local (también llamado «puerta de enlace»). Este dato también lo debe proporcionar el administrador de la red. Supongamos para este ejemplo que esa dirección es `20.3.4.1`. Puedes crear la entrada en la tabla de encaminamiento con el siguiente comando:

```
$ sudo ip route add default via 20.3.4.1
$ sudo ip route
default via 20.3.4.1 dev eno1
20.3.4.0/24 dev eno1 proto kernel scope link src 20.3.4.13
```

Esto corresponde con la tabla de encaminamiento básica que vimos en §8.3.2. La tabla dice cómo llegar a los vecinos y cómo llegar a cualquier otro nodo fuera de la red. Por eso se define como router por defecto.

Si no haces nada más, esta información se perderá cuando el nodo se reinicie. Para que sea persistente, tienes que escribirla en el archivo de configuración `/etc/network/interfaces`, que quedaría así:


```
auto eno1
iface eno1 inet static
    address 20.3.4.13
    netmask 255.255.255.0
    gateway 20.3.4.1
```

Este es el método clásico de Debian, pero existen otros métodos como `systemd-networkd`, `NetworkManager` o `netplan`, que aunque más versátiles, para un uso tan básico como este, resultan equivalentes.

Otra cuestión importante es el nombre del nodo. Se utiliza para identificar el nodo en la red y puede ser necesario para configurar ciertos servicios. El nombre del nodo se puede consultar y cambiar con el comando `hostname`:

```
$ hostname
maine
$ hostname atlantis
$ hostname
atlantis
```

Pero también es temporal. Si quieres que se mantenga después de reiniciar debes escribir ese nombre en el archivo `/etc/hostname`.

Este tipo de configuración se suele denominar «manual» o «estática» porque el usuario la tiene que definir explícitamente y no cambia a menos que lo haga el usuario. Cualquier SO (incluidos los móviles) tiene algún medio para configurar esta información de forma manual, ya sea con una shell o con una interfaz gráfica.

10.2. Configuración automática con DHCP

Pedir a un usuario sin conocimientos técnicos básicos que realice una configuración manual no es lo más conveniente. Además cada nuevo usuario para cada nuevo nodo u otro equipo como una impresora, debe solicitar la configuración al administrador de la red. El administrador a su vez debe llevar registro de las direcciones asignadas por cada subred, y cerciorarse de que se siguen usando, pues lo más probable es que un usuario que se va olvide notificar que ya no necesita esas direcciones.

Para resolver todos estos inconvenientes existe el protocolo DHCP (Dynamic Host Configuration Protocol) [22]. DHCP es un servicio que proporciona la configuración completa para un nodo, cuando éste lo solicita. DHCP se suele considerar un protocolo de aplicación que se encapsula sobre UDP y utiliza los puertos reservados 67 y 68 para servidor y cliente respectivamente.

Pero, ¿cómo puede un nodo sin dirección solicitar algo a un servidor? ¿Cómo sabe el nodo la dirección del servidor? Es posible porque el cliente

DHCP envía mensajes de difusión (dirigidos a 255.255.255.255), algo que es posible incluso aunque la interfaz no tenga una dirección asignada.

En realidad, la conversación entre cliente y servidor DHCP es algo más compleja. Veamos el escenario típico con más detalle.

- El cliente, en el nodo que necesita configuración, envía un mensaje Discover a la dirección de difusión 255.255.255.255. El propósito de este mensaje es «descubrir» servidores DHCP en la red.
- El servidor (o servidores) DHCP recibe esos mensajes y en función de su configuración responde con un mensaje Offer que contiene la dirección IP y otros parámetros de configuración. Este mensaje también va dirigido a la dirección de difusión.
- El cliente recibe estas «ofertas» y envía un mensaje Request para confirmar la que ha elegido.
- El servidor envía un mensaje Ack como confirmación. Este mensaje contiene la información de configuración completa, incluyendo la dirección IP, la máscara de subred, la puerta de enlace y también los servidores DNS. Desde este momento, el servidor considera que la dirección IP está asignada a ese cliente y no la ofrecerá a otros.
- Al recibir el mensaje Ack, el cliente aplica la configuración a la interfaz de red, y a partir de ese momento puede comunicarse con normalidad.
- Cuando el nodo se apaga o ya no necesita la dirección, envía un mensaje Release para liberarla. Así el servidor puede disponer de ella y asignarla a otro nodo.

Puedes hacer una captura de una conversación DHCP real. En este ejemplo se utiliza `eno1`, que es una interfaz Ethernet, pero lo puedes probar igualmente con una interfaz WiFi. En un terminal arranca una captura con `tshark -i eno1 -Y bootp`. El argumento `-Y bootp` es un filtro que le dice a `tshark` que te interesan solo los mensajes BOOTP, que incluye los mensajes DHCP, porque el segundo es una extensión del primero.

En otro terminal, ejecuta el cliente tal como aparece a continuación. Lo más probable es que ya tengas el programa `dhclient`, pero si no es así, instala el paquete `isc-dhcp-client`.

El siguiente comando libera la dirección que tienes asignada (envía el mensaje Release) para que así puedas capturar el proceso completo. Opcionalmente después de ese comando, puedes ver que ahora la interfaz no tiene dirección si ejecutas `ip a`.

```

1 $ sudo dhclient -v -r eno1
2 Killed old client process
3 Internet Systems Consortium DHCP Client 4.4.3-P1
4
5 Listening on LPF/eno1/f8:5f:2a:c0:ff:ee
6 Sending on   LPF/eno1/f8:5f:2a:c0:ff:ee
7 DHCPRELEASE of 172.24.27.53 on eno1 to 172.20.32.167 port 67

```

Este segundo comando es el que realiza la solicitud. Puedes ver en la propia salida del comando un resumen de los mensajes de los que hemos hablado.

```

$ sudo dhclient -v -i eno1
Internet Systems Consortium DHCP Client 4.4.3-P1

Listening on LPF/eno1/f8:5f:2a:c0:ff:ee
Sending on   LPF/eno1/f8:5f:2a:c0:ff:ee
DHCPDISCOVER on eno1 to 255.255.255.255 port 67 interval 5
DHCPDISCOVER on eno1 to 255.255.255.255 port 67 interval 10
DHCPOFFER of 172.24.27.53 from 172.20.32.167
DHCPREQUEST for 172.24.27.53 on eno1 to 255.255.255.255 port 67
DHCPACK of 172.24.27.53 from 172.20.32.167
bound to 172.24.27.53 -- renewal in 1571 seconds.

```

En la captura que tenías en marcha en el otro terminal ha debido aparecer algo similar a esto:

```

$ sudo tshark -i eno1 -Y bootp
1      0.0.0.0 → 255.255.255.255 DHCP 342 DHCP Discover - Transaction ID 0x9b6d7a5a
2 172.20.32.167 → 172.24.27.53   DHCP 342 DHCP Offer   - Transaction ID 0x9b6d7a5a
3      0.0.0.0 → 255.255.255.255 DHCP 342 DHCP Request - Transaction ID 0x9b6d7a5a
4 172.20.32.167 → 172.24.27.53   DHCP 342 DHCP ACK    - Transaction ID 0x9b6d7a5a

```

Para ver más detalles, repite la captura añadiendo el argumento `-v` al comando `tshark`. Se muestran aquí los campos más interesantes de los mensajes Discover y Offer.

```

$ sudo tshark -i eno1 -Y bootp -v
Dynamic Host Configuration Protocol (Discover)
  Message type: Boot Request (1)
  Hardware type: Ethernet (0x01)
  Hardware address length: 6
  Transaction ID: 0x014ff92a
  Client IP address: 0.0.0.0
  Your (client) IP address: 0.0.0.0
  Next server IP address: 0.0.0.0
  Relay agent IP address: 0.0.0.0
  Client MAC address: f8:5f:2a:c0:ff:ee
  Option: (50) Requested IP Address (0.0.0.0)
  Option: (12) Host Name
    Length: 5
    Host Name: maine
  Option: (55) Parameter Request List
    Length: 13
    Parameter Request List Item: (1) Subnet Mask

```

```

Parameter Request List Item: (28) Broadcast Address
Parameter Request List Item: (2) Time Offset
Parameter Request List Item: (3) Router
Parameter Request List Item: (15) Domain Name
Parameter Request List Item: (6) Domain Name Server
Parameter Request List Item: (119) Domain Search
Parameter Request List Item: (12) Host Name
Parameter Request List Item: (26) Interface MTU
Parameter Request List Item: (121) Classless Static Route
Parameter Request List Item: (42) Network Time Protocol Servers
Option: (61) Client identifier
  Length: 19
  IAID: 96a502be
  DUID Type: link-layer address plus time (1)
  Hardware type: Ethernet (1)
  Time: 799853663
  Link layer address: f8:5f:2a:c0:ff:ee

Dynamic Host Configuration Protocol (Offer)
  Message type: Boot Reply (2)
  Hardware type: Ethernet (0x01)
  Hardware address length: 6
  Transaction ID: 0x014ff92a
  Client IP address: 0.0.0.0
  Your (client) IP address: 172.24.27.53
  Next server IP address: 172.20.32.167
  Client MAC address: f8:5f:2a:c0:ff:ee
  Option: (1) Subnet Mask (255.255.255.0)
  Option: (58) Renewal Time Value: 30 minutes (1800)
  Option: (59) Rebinding Time Value: 52 minutes, 30 seconds (3150)
  Option: (51) IP Address Lease Time: 1 hour (3600)
  Option: (54) DHCP Server Identifier (172.20.32.167)
  Option: (3) Router: 172.24.27.1
  Option: (15) Domain Name: uclm.es
  Option: (6) Domain Name Server:
    Domain Name Server: 172.20.32.3
    Domain Name Server: 172.20.32.4

```

En el mensaje Discover el cliente solicita diversa información. Lo puedes ver en la sección *Parameter Request List*. El servidor responde a estas solicitudes como opciones al final del mensaje Offer y Ack. No tiene por qué responder a todo, pero a parte por supuesto de la dirección del nodo (campo «Your (client) IP address»), normalmente siempre aparecerán los ya nombrados:

- Subnet Mask
- Router
- Domain Name Server

El servidor asigna una dirección al cliente por un período limitado: el «tiempo de concesión» (*lease time*), que es configurable en el servidor. En la captura esto aparece en el campo *Rebinding Time Value*. Durante ese plazo, el servidor asegura que la dirección no se va a asignar a otro cliente. Si el cliente quiere seguir utilizando la misma dirección puede enviar un

mensaje Renew para renovar la concesión antes de que termine el plazo de renovación (campo **Renewal Time Value** en la captura). En la captura puedes comprobar que el tiempo de concesión es de 52 min 30 seg y el de renovación es de 30 min. Estos valores son bastante bajos porque hemos tomado la captura en un entorno empresarial. En el entorno doméstico el tiempo de concesión suele ser de 24 horas o incluso más. Comprueba qué valores aparecen en tu caso analizando tu captura.

10.3. Servidor DHCP

El programa `dnsmasq` es un servidor DHCP sencillo, ligero y fácil de configurar, ideal para redes domésticas, despliegues pequeños o para realizar pruebas de laboratorio. También funciona como servidor DNS aunque para este ejemplo lo vamos a desactivar.

El servidor se configura mediante un archivo de texto que suele estar en `/etc/dnsmasq.conf`. El siguiente listado es un ejemplo perfectamente funcional de ese fichero que sirve para asignar direcciones en la red `192.168.2.0/24` incluyendo la información básica necesaria.

```
port=0
listen-address=192.168.2.1
dhcp-range=192.168.2.2,192.168.2.100,12h
dhcp-option=3,192.168.2.1
dhcp-option=6,8.8.8.8,1.1.1.1
```

Veámos en detalle su sintaxis y significado:

port es el puerto en el que debe escuchar el servidor DNS. Al asignar un valor de 0, el servidor DNS queda inoperante.

listen-address indica la dirección de la interfaz de red en la que el servidor DHCP atenderá peticiones.

dhcp-range es el rango de direcciones que puede asignar a los clientes. Desde `192.168.2.2` hasta `192.168.2.100`, con una concesión de 12 horas. El nodo que ejecuta este servidor DHCP probablemente actúa de router de la LAN (aunque no es obligatorio) y tendría la dirección `192.168.2.1`.

dhcp-option indica distintas opciones con información adicional que se puede ofrecer a los clientes. Los números de opción son los de la especificación de DHCP y corresponden con la captura que has visto en la sección anterior: 3 para indicar la dirección del router y 6 para las direcciones IP de dos servidores DNS, que en este caso son servidores públicos de Google y Cloudflare.

Puedes verlo en funcionamiento aplicando este mismo fichero de configuración con el ejemplo /dnsmasq utilizando un contenedor appdocker como puedes ver en la siguiente consola:

```
dnsmasq$ docker compose up
[+] Running 1/1
- Container dnsmasq Create
Attaching to dnsmasq
dnsmasq | dnsmasq[1]: started, version 2.90 DNS disabled
dnsmasq | dnsmasq-dhcp[1]: DHCP, IP range 192.168.2.2 -- 192.168.2.100, lease time 12h
```

En este ejemplo `docker` crea una red llamada `dnsmasq_net` en la que el host (tu PC) y el contenedor son vecinos. Eso significa que puedes solicitar una dirección a este DHCP. Debes listar las interfaces para averiguar cuál tiene una IP en el bloque `192.168.2.0/24`. Sí, la interfaz ya tiene una IP asignada, pero no importa, aún así puedes solicitar otra al servidor DHCP y comprobar su funcionamiento. La interfaz en cuestión tendrá un nombre similar a `br-f215a045cd10`. Por tanto, en otro terminal, puedes solicitar una dirección con:

```
$ ip a
[...]
54: br-f215a045cd10: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP
    link/ether 5a:47:ab:01:41:42 brd ff:ff:ff:ff:ff:ff
    inet 192.168.2.254/24 brd 192.168.2.255 scope global br-f215a045cd10
        valid_lft forever preferred_lft forever

$ sudo dhclient -v -i br-f215a045cd10
```

Al ejecutar esto deberías ver un registro detallado de las solicitudes y respuestas en la salida de `docker compose` con información similar a la captura `tshark` de la sección anterior. Después puede comprobar que la interfaz tiene ahora una dirección del rango (`192.168.2.85`).

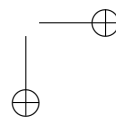
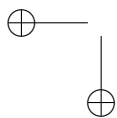
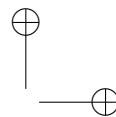
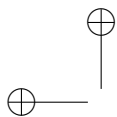
```
$ ip addr show br-f215a045cd10
54: br-f215a045cd10: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP
    link/ether 5a:47:ab:01:41:42 brd ff:ff:ff:ff:ff:ff
    inet 192.168.2.254/24 brd 192.168.2.255 scope global br-f215a045cd10
        valid_lft forever preferred_lft forever
    inet 192.168.2.85/24 brd 192.168.2.255 scope global secondary dynamic br-f215a045cd10
        valid_lft 43196sec preferred_lft 43196sec
```

Y ¿qué más?

La dirección IP del nodo, la del router y las de los servidores DNS constituyen la configuración mínima esencial para que un nodo pueda conectarse a la red, sin embargo hay mucho más. Conforme una red y un despliegue TIC se vuelve más y más complejo es necesario administrar muchos otros

parámetros como inventarios de equipos, usuarios, programas instalados, versiones y un largo etcetera.

Esto da lugar a toda una disciplina que es la «Gestión y administración de redes y sistemas» en sus muchas variantes. En esta disciplina se utilizan protocolos y herramientas específicas. Y muy relacionado con todo esto quedan las tareas de monitorización y seguridad, que suponen otra gran área de conocimiento y estudio. Este capítulo solo ha sido el primer paso hacia ese mundo.



Capítulo 11

Confiabilidad y control de flujo

Al terminar este capítulo, entenderás:

- Qué es la confiabilidad en las comunicaciones.
- Qué es el control de flujo y por qué es necesario para lograr confiabilidad.
- Qué relación existe entre confiabilidad y control de flujo.
- Qué son y cómo funcionan los protocolos ARQ.
- En qué consiste y cómo funciona la ventana deslizante.

Que los datos lleguen de forma fiable a su destino se requiere al menos dos mecanismos básicos: confiabilidad y control de flujo. Ambos se pueden implementar por protocolos de capas diferentes, y se pueden lograr con enfoques dispares, sin embargo lo más habitual es encontrarlos en la capa de transporte (como TCP) o en la de enlace (como HDLC o PPP).

Decimos que una comunicación es fiable o **confiable** si los datos llegan a su destino exactamente tal cual partieron del origen, sin ningún cambio ni omisión, pero también sin partes duplicadas o desordenadas. Hay muchas razones por las que podrían darse todos esos problemas: interferencias, atenuación, retrasos, disparidad de rutas, limitación en el almacenamiento intermedio, etc.

Por otro lado, el **control de flujo** es un mecanismo que permite a los participantes limitar o interrumpir temporalmente el flujo de datos entre ellos. Aunque el concepto se puede aplicar de forma genérica para varias situaciones, aquí emplearemos la expresión «control de flujo» para un caso específico: el ajuste que el receptor solicita al emisor para evitar que le saturé.

La razón por la que estudiamos la confiabilidad y el control de flujo a la vez se debe a que, habitualmente, ambos se diseñan e implementan mediante un mismo mecanismo. En todo caso, el control de flujo es imprescindible

para lograr confiabilidad. Sin control de flujo, el receptor perdería datos por saturación.

Todo mensaje que viaja entre dos dispositivos lo hace a través de algún medio físico, sea el aire, un cable de cobre o una fibra óptica, y por eso está restringido por las limitaciones físicas. Incluso en el vacío, la señal sufre cambios causados por interferencias o ruido electromagnético, y la intensidad de la señal se degrada por la distancia o la resistencia del medio. Por eso, cualquier mensaje que se envíe a través de una red podría llegar modificado o incompleto, o incluso no llegar. Llamamos a todo esto «errores de transmisión» y para solucionarlos existen distintas técnicas, que se clasifican en dos categorías: detección y corrección de errores.

La detección de errores más simple consiste en añadir información redundante al mensaje original, normalmente solo unos pocos bytes. Puede ser algo tan sencillo como un bit de paridad, pero típicamente se utiliza un polinomio de redundancia cíclica (CRC). El emisor aplica ese algoritmo a los datos de partida y adjunta el resultado al mensaje. Al llegar, el receptor realiza el mismo cálculo y compara el resultado con el que ha recibido. Si estos valores coinciden, el mensaje es correcto; si difieren, el mensaje ha sufrido algún cambio.

En la mayoría de los casos, estos algoritmos solo permiten detectar la alteración, pero no pueden determinar qué parte del mensaje es la que ha sido alterada y menos aún cuál fue la alteración. Este es el caso del FCS de Ethernet, o del checksum utilizado en las cabeceras de IP, UDP, TCP, etc., que es un valor de tan solo 16 bits. Puedes ver una explicación detallada de este algoritmo en [🔗/inet-checksum/checksum.ipynb](#).

Si la probabilidad de error es baja, lo más sencillo y eficiente es detectar los errores y retransmitir los mensajes afectados. Si por el contrario la probabilidad es alta, o no es posible o conveniente realizar la retransmisión, entonces se utilizan algoritmos de corrección de errores. Estos algoritmos añaden una cantidad significativa de información redundante a los mensajes y también requieren recursos de cómputo adicionales. El más conocido de estos algoritmos de corrección es Reed-Solomon, que se utiliza en comunicaciones vía satélite.

Para la retransmisión de mensajes erróneos se utilizan variantes de los protocolos ARQ (Automatic Repeat reQuest) en los que el receptor debe enviar mensajes de **confirmación** (también llamados «reconocimientos» o ACK). Estos protocolos permiten al emisor obtener información actualizada sobre el estado del receptor. Cada vez que el emisor recibe una confirmación tiene constancia de que el receptor está listo para continuar, y envía un nuevo

mensaje, es decir, al mismo tiempo proporciona confirmación y controla el flujo.

Aunque se denominan «protocolos», los ARQ no definen un formato concreto de mensaje o una sintaxis precisa. Únicamente definen una serie de reglas y pautas que deben cumplir los mensajes de datos y confirmación. Podríamos entenderlos como una especie de patrones que se pueden utilizar para crear protocolos confiables concretos.

Veamos como funciona cada uno de ellos.

11.1. Parada y espera

En el protocolo *parada y espera* (*stop and wait*) el emisor envía un mensaje y espera su confirmación. Si no la recibe en un tiempo determinado, reenvía el mensaje. Si recibe la confirmación, envía el siguiente mensaje.

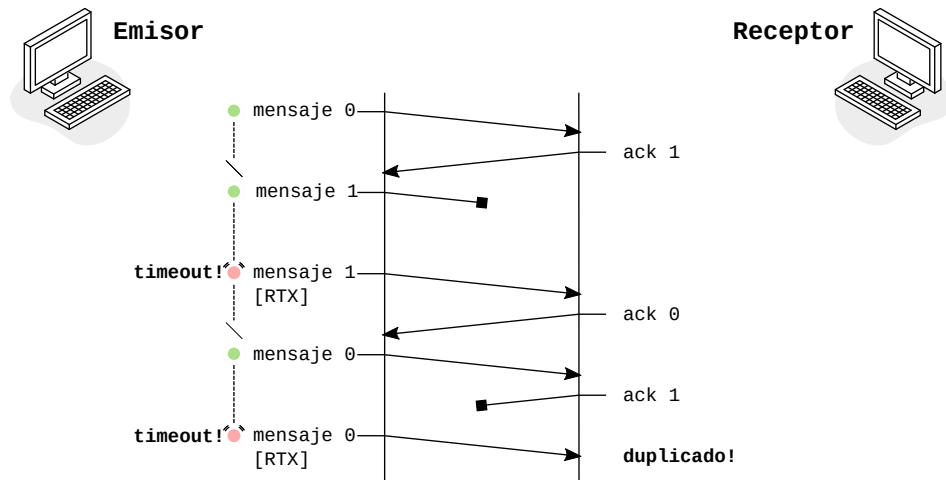
En detalle:

- El emisor envía un mensaje de datos que incluye un campo checksum.
- El emisor inicia un temporizador de retransmisión.
- El receptor recibe el mensaje y calcula el checksum.
 - Si el checksum coincide, envía una confirmación.
 - Si el checksum no coincide, descarta el mensaje.
- Si el emisor recibe la confirmación, descarta el temporizador y envía el siguiente mensaje.
- Si el temporizador expira antes de recibir la confirmación, reenvía el mensaje.

Si el tiempo de propagación del mensaje es variable e impredecible o el temporizador no está bien ajustado, puede ocurrir que expire antes de que la confirmación llegue al emisor. Si eso ocurre, el emisor envía una retransmisión prematura¹ y el receptor recibirá un duplicado del mensaje. Para que el receptor pueda detectar esta situación se utiliza un número de secuencia de un único bit: alterna entre 0 y 1. Si el receptor recibe dos mensajes consecutivos con el mismo número de secuencia, sabrá que es un duplicado, y lo descartará.

Respecto a las confirmaciones, hay un detalle que puede causar confusión y merece la pena aclarar. Es muy frecuente que muchas implementaciones tomen el convenio de enviar en la confirmación el número de secuencia del **siguiente** mensaje esperado en lugar del que corresponde al mensaje que acaba de llegar. Es decir, para confirmar el mensaje de datos con número de

¹llamada a menudo «retransmisión espuria»

FIGURA 11.1: Diagrama de secuencia del protocolo *parada y espera*

secuencia 0, el receptor envía una confirmación con el número de secuencia 1. En este texto también tomaremos este convenio.

La implementación de *parada y espera* es simple, pero ineficiente. Como el emisor debe esperar la confirmación antes de poder enviar el siguiente mensaje, el canal de comunicación se infrautiliza en gran medida, afectando gravemente a la velocidad de transmisión neta.

11.2. Repetición continua

Repetición continua (go back N) intenta maximizar la ocupación del canal. Para ello, el emisor envía varios mensajes consecutivos, sin esperar una confirmación para enviar el siguiente. Esa técnica se llama *canalización (pipelining)* y consigue una mejora sustancial del ancho de banda efectivo.

Para que funcione, el emisor tiene que guardar temporalmente los mensajes no confirmados por si fuera necesario retransmitirlos. El espacio donde se almacenan estos mensajes suele ser una cola circular que se conoce como *buffer de envío*.

Cuando el receptor recibe un mensaje, comprueba el *checksum*, envía una confirmación al emisor y entrega el mensaje a la capa superior. Si algún mensaje se pierde, el receptor recibe mensajes fuera de secuencia. Si eso pasa, los descarta aunque sean correctos y, para cada uno de ellos, envía obligatoriamente una confirmación para el último mensaje recibido correctamente (y que respetaba la secuencia sin huecos), es decir, el emisor puede

recibir múltiples confirmaciones para un mismo mensaje de datos. Se las llama «confirmaciones duplicadas».

Los mensajes incluyen también un número de secuencia que el receptor utiliza para confirmarlos. El espacio de número de secuencia suele ser potencia de 2 porque se representa en un *campo de n bits* en la cabecera del mensaje. De este modo, con n bits se obtiene un espacio de números de secuencia de 2^n mensajes, si bien no es posible enviar n mensajes sin confirmación, porque provoca una situación ambigua que hay que evitar. Imagina una implementación hipotética con un número de secuencia de 2 bits (2^2 mensajes) en la que se utilizaran todos los números disponibles:

- El emisor envía los mensajes con números de secuencia 0 a 3 incluido.
- Todos los mensajes llegan correctamente a su destino
- Todas las confirmaciones se pierden, pero el receptor lo desconoce.
- El temporizador del mensaje 0 expira.
- El emisor retransmite el mensaje 0.

En esa situación el receptor tomaría esa retransmisión por un nuevo mensaje que está reutilizando el número de secuencia 0, provocando el fallo del protocolo. Para evitar esta situación, se define una *ventana de emisión* cuyo tamaño debe ser menor que 2^n , normalmente:

$$\text{tamaño de ventana} \leq 2^n - 1 \quad (11.1)$$

La ventana determina qué mensajes puede enviar el emisor en un momento dado, sin esperar confirmaciones y sin provocar ambigüedad. Cuando un mensaje es confirmado, su número de secuencia queda libre para ser reutilizado con un nuevo mensaje y el buffer se reutiliza una y otra vez durante la transmisión. Esa reutilización cíclica de los números de secuencia se denomina *ventana deslizante* (*sliding window*).

En la Figura 11.2 puedes cómo se evita la ambigüedad. El mensaje con número de secuencia 3 no forma parte de la ventana (marcada con el recuadro gris) hasta que el al menos mensaje 0 sea reconocido. El ACK 1, confirmando el mensaje 0, es el que provoca el deslizamiento de la ventana, y abre la posibilidad de enviar el mensaje 3.

Normalmente estos protocolos utilizan confirmación *acumulativa*, es decir, que incluye también a los mensajes anteriores. Por ejemplo, una confirmación acumulativa con número de secuencia 4 significa que los mensajes hasta el 3 incluido han llegado correctamente, y no hay necesidad de enviar previamente una confirmación individual para cada uno de los números de secuencia previos. Esto permite que algún mensaje de confirmación se pue-

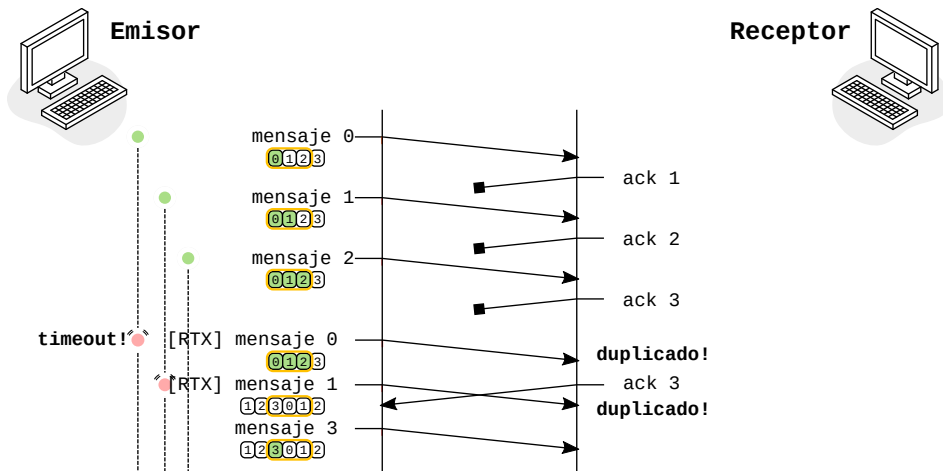


FIGURA 11.2: Diagrama de secuencia del protocolo *repetición continua* con confirmación acumulativa

da perder (o no enviarse) sin consecuencias. En realidad esto aumenta la eficiencia y muchos protocolos retrasan intencionadamente la confirmación (se llama «ACK retardado» o *delayed ACK*) para poder enviar una única confirmación para varios mensajes de datos.

Nótese en todo caso que el receptor no establece temporizadores ni retransmisiones cuando envía confirmaciones, y por supuesto el emisor no confirma las confirmaciones que recibe.

El nombre *go back N* hace referencia al hecho de que en caso de perder un mensaje quizá haya que «volver N mensajes atrás» y reenviarlos.

11.3. Repetición selectiva

Con *repetición selectiva* (*selective repeat*), el emisor también envía varios mensajes de forma consecutiva y espera las confirmaciones, pero a diferencia de *repetición continua*, si un mensaje se pierde o corrompe, el emisor reenvía solo el mensaje afectado, pero no los siguientes, y eso da nombre del protocolo.

Cuando se pierde un mensaje y el receptor empieza a recibir mensajes fuera de secuencia, aparte de confirmar el último mensaje correcto y en orden, ahora en lugar de descartarlos, los almacena en un *buffer de recepción*. Cuando los temporizadores de los mensajes no confirmados van expirando, el emisor realiza las correspondientes retransmisiones, pero tan pronto como el receptor «rellene los huecos» enviará una confirmación para todo

el conjunto y los temporizadores pendientes se cancelarán evitando más retransmisiones innecesarias.

En la Figura 11.3 puedes ver un ejemplo: Supongamos una ventana de 3 bits, en la que un emisor ha enviados los mensajes [0-4] perdiéndose el 2. El receptor enviará una confirmaciones para los mensajes con secuencia 1 y 2, reconociendo respectivamente los mensajes 0 y 1. Fíjate que al disponer de confirmación acumulativa, podría no enviar la primera de esas confirmaciones. Cuando lleguen los mensajes 3 y 4 los almacenará en su buffer de recepción y enviará sendas confirmaciones con secuencia 2. Más tarde, cuando expire el temporizador del mensaje 2, será retransmitido. El receptor al recibirlo dispondrá de la secuencia completa [0-4] y enviará un ACK con secuencia 5 para confirmar todos los mensajes recibidos. Mientras tanto es posible que más temporizadores expiren (el mensaje 3 en la figura) con lo que habrá retransmisiones innecesarias. Al llegar el ACK 5 se confirman todos los mensajes pendientes y se cancelan sus temporizadores (la figura solo muestra los de los mensajes [2-4] que son los relevantes para el ejemplo).

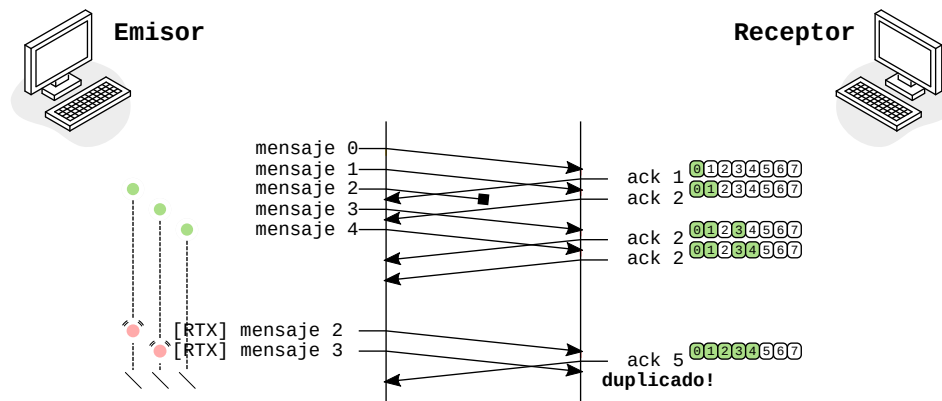


FIGURA 11.3: Diagrama de secuencia del protocolo *repetición selectiva*

Es posible mejorar también esta situación, aunque como siempre, a cambio de complejidad adicional. El receptor puede utilizar un nuevo tipo de confirmación —la confirmación negativa o NACK— para notificar explícitamente cuál es el mensaje corrupto o perdido. De ese modo, el emisor puede retransmitir inmediatamente ese mensaje en lugar de esperar a que expire su temporizador, y de ese modo aprovechando mejor el canal.

11.4. Confiabilidad en comunicaciones duplex

Los protocolos de confiabilidad que hemos visto tratan la comunicación solo en un sentido de la comunicación. Hemos hablado todo el tiempo de «emisor» y «receptor». En realidad se explica de este modo por simplicidad y para ayudar a entender su funcionamiento. Nada impide que en un canal duplex ambos extremos actúen como emisores y receptores a la vez. En ese caso, los protocolos que hemos visto trabajan de forma independiente para cada uno de los sentidos de la comunicación empleando ventanas y números de secuencia independientes.

En realidad, utilizar protocolos de confiabilidad en ambos sentidos permite optimizar algunas situaciones. Por ejemplo, resulta ineficiente enviar varios mensajes pequeños consecutivos. Por eso, cuando uno de los extremos necesita enviar una confirmación, que solo ocupa una simple cabecera, puede esperar «un poco» y aprovechar un mensaje de datos saliente para colocar la confirmación en su cabecera. A este tipo de optimizaciones se las llama *piggybacking* y es tan habitual que los formatos de las cabeceras se diseñan específicamente para hacerlo posible.

Y ¿qué más?

Los protocolos ARQ son la base de la mayoría de los protocolos que proporcionan confiabilidad en distintas capas. Algunos de estos protocolos son IEEE 802.11 (Wi-Fi), HDLC, Bluetooth o LTE, que son protocolos de la capa de enlace², TCP o SCTP que son protocolos de transporte, o TFTP que es un protocolo de aplicación. Sin duda alguna, el más importante de todos estos protocolos es TCP. Por eso trataremos en detalle su funcionamiento y características en el capítulo 12.

²En realidad, algunos de estos protocolos abarcan varias capas, pero los mecanismos de confiabilidad los ofrecen en la capa de enlace.

Capítulo 12

Confiabilidad y control de flujo en TCP

Al terminar este capítulo, entenderás:

- Cómo se establece y termina una conexión TCP.
- Qué implicaciones tiene el control de flujo de TCP.
- Cómo detecta y resuelve TCP los errores de transmisión.

Ya hemos hablado del protocolo de transporte TCP. Aquí veremos el soporte que ofrece para confiabilidad y control de flujo [23]. TCP utiliza un mecanismo de retransmisión de mensajes y confirmaciones de tipo repetición selectiva con confirmación acumulativa y ventana deslizante. Es *full duplex* de modo que aplica lo dicho en § 11.4. También puede utilizar un sistema de confirmación selectiva para mejorar la eficiencia, aunque es algo opcional que deben negociar explícitamente los extremos durante el establecimiento de la conexión.

Para cada conexión establecida, el SO crea un TCB (Transmission Control Block). Se trata de una estructura de datos que almacena toda la información relacionada con la conexión: dirección del otro extremo, números de secuencia, temporizadores, puertos, estado, etc. y también el buffer dónde coloca los datos que el proceso emisor envía —`socket.send()`—, llamado *buffer de envío* (*sending buffer*). En el otro extremo, el receptor tiene un *buffer de recepción* (*receiving buffer*) dónde almacena los datos hasta que el proceso receptor los solicite, con `socket.recv()`.

A partir de los datos almacenados en el buffer de envío, el procedimiento de **segmentación**, determina cuántos bytes se incluirán en el siguiente mensaje TCP (llamado «segmento») que se envíe. El tamaño del segmento depende de varias variables que veremos en las siguientes secciones y también cuando hablemos de congestión (§ 13.5).

Una diferencia interesante respecto a los protocolos de estilo ARQ es que TCP no identifica los segmentos. En lugar de ello, numera cada uno de los bytes del flujo completo, para lo que usa un entero de 32 bits. En particular, el número de secuencia que corresponde al primer byte de la carga útil de cada segmento aparece en el campo *sequence number* de su cabecera y es por eso que se le llama «número de secuencia del segmento». Si por ejemplo se envía un segmento con número de secuencia 1000 y su carga útil tiene 500 bytes, el número de secuencia del siguiente byte (que será el primero del siguiente segmento) será 1500. Del mismo modo, la confirmación que envía el receptor corresponde al número de secuencia del siguiente byte que espera recibir, y que aparecerá en el campo *acknowledge number* de la cabecera.

12.1. Conexión

El procedimiento de conexión de TCP permite a los dos extremos intercambiar varios parámetros de configuración. El más importante es el *número de secuencia inicial* de cada extremo (o ISN). Cada uno de los extremos genera su ISN mediante una función pseudo-aleatoria, que aparecerá únicamente en el primer segmento que envíen. A partir de ese valor, se numera cada byte del flujo de datos. La conexión responde a un patrón muy concreto de tres mensajes, llamado *three-way handshake*, que se suele traducir literalmente como *triple apretón de manos*.

```
1. --> seq:1200, SYN
2 <-- seq:4800, SYN/ACK, ack:1201
3. --> seq:1201, ACK, ack:4801
```

LISTADO 12.1: TCP: Secuencia de una conexión

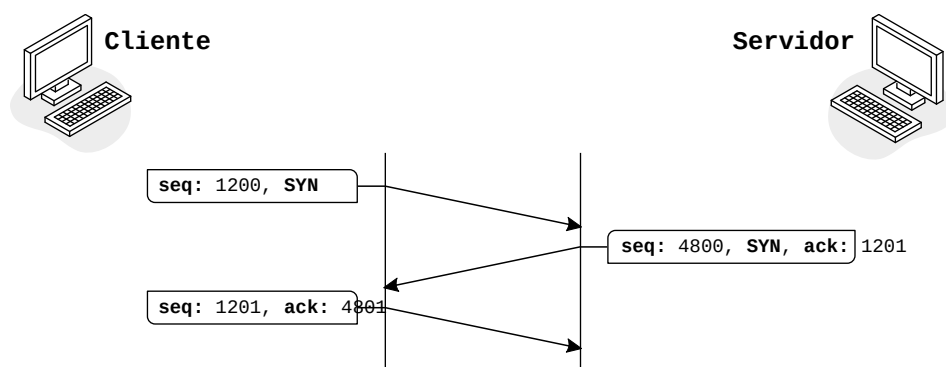


FIGURA 12.1: TCP: Patrón *triple handshake* durante la conexión

La conexión la inicia siempre el cliente, por lo que el primer segmento es suyo. Este segmento incluye el ISN del cliente (es el 1200 del ejemplo de la Figura 12.1 y del esquema equivalente del Listado 12.1) y lleva activo el flag SYN¹ únicamente. El servidor responde con un segmento que incluye su ISN (4800 en la figura) y lleva activos los flags SYN y ACK, y por tanto el contenido del campo *acknowledge number* es relevante y sirve como confirmación del primer mensaje del cliente (1201). En realidad el flag ACK ya siempre estará activo hasta el fin de la conexión. El segundo segmento del cliente lleva el ACK 4801, reconociendo así el segmento del servidor. El número de secuencia de este tercer segmento es 1201 a pesar de que el cliente no ha enviado aún ningún byte de datos. Esto ocurre tanto en la conexión como en la desconexión y es una excepción al modo en que se incrementan los números de secuencia. A partir de ese instante, la conexión queda establecida y ambos extremos conocen los números de secuencia que va a utilizar el otro, es decir, los números de secuencia están sincronizados (*SYNchronized*).

Puedes ver la captura de la conexión real en la Figura 12.2. El comando ejecuta `tshark` en segundo plano, espera 1 segundo y luego conecta al puerto 80 del servidor `example.net` con `ncat`. Si lo pruebas en tu computador, recuerda parar `tshark` al terminar, por ejemplo con `kill%1`².

```
$ sudo tshark -f "tcp port 80" & sleep 1; ncat example.net 80
1 192.168.0.37 → 93.184.215.14 TCP 74 50192 → 80 [SYN] Seq=0 Win=32120 Len=0
    MSS=1460 SACK_PERM WS=128
2 93.184.215.14 → 192.168.0.37 TCP 74 80 → 50192 [SYN, ACK] Seq=0 Ack=1 Win=65535 Len=0
    MSS=1452 SACK_PERM WS=512
3 192.168.0.37 → 93.184.215.14 TCP 66 50192 → 80 [ACK] Seq=1 Ack=1 Win=32128 Len=0
```

LISTADO 12.2: TCP: Captura de una conexión

La captura indica tanto para el cliente como para el servidor que los ISN son 0 (Seq=0), pero cuidado, porque esto no es real. Para poder hacer un seguimiento más sencillo de los números de secuencia, `tshark`³ no muestra el número de secuencia que aparece en el segmento, si no un número relativo al ISN; por eso siempre es 0 en el primer segmento. Hay una opción para forzar a `tshark` a mostrar los números de secuencia reales. Puedes ver el

¹por *synchronize*

²El comando `kill` envía una señal (por defecto SIGTERM) al proceso indicado, en este caso con la intención de terminarlo. En el capítulo 2 vimos el uso de este y otro muchos comandos y recuerda también que el anexo A es un pequeño catálogo de comandos habituales. Y por supuesto, siempre puedes usar el comando `man` para obtener información de cualquier comando.

³Y la mayoría de los *sniffers*

mismo ejemplo con números de secuencia reales en la captura 12.3. Se han omitido algunos campos que no son relevantes para este ejemplo.

```
$ sudo tshark -f "tcp port 80" -o tcp.relative_sequence_numbers:FALSE &
$ sleep 1; ncat example.net 80
1 192.168.0.37 → 93.184.215.14 TCP 74 48200 → 80 [SYN] Seq=7967281 Win=32120 Len=0
2 93.184.215.14 → 192.168.0.37 TCP 74 80 → 48200 [SYN, ACK] Seq=6696166 Ack=7967282 Len=0
3 192.168.0.37 → 93.184.215.14 TCP 66 48200 → 80 [ACK] Seq=7967282 Ack=6696167 Len=0
```

LISTADO 12.3: TCP: Captura de una conexión (con números de secuencia reales)

12.2. Tamaño máximo de segmento

La especificación de TCP dice que todo computador debe ser capaz de recibir segmentos de como mínimo 536 bytes en IPv4 y 1220 en IPv6 [23]. Este requisito está relacionado con la capacidad de almacenamiento en el *buffer de recepción*. Sin embargo, si el emisor tiene la certeza de que el receptor puede manejar segmentos mayores, es preferible que lo haga ya que eso aumenta el rendimiento del canal.

En general es más eficiente enviar pocos mensajes grandes que muchos pequeños. Enviando menos mensajes se reducen las interacciones entre capas a través de las interfaces, se reduce el peso relativo de las cabeceras en el total de datos transmitidos —la *sobrecarga de cabeceras*—, se minimiza el impacto de las retransmisiones y el trabajo que conlleva para routers y conmutadores.

Sin embargo, hay un límite superior relacionado con el tamaño máximo de datagrama IP, que a su vez está limitado por la MTU de la red. Si el tamaño del segmento elegido provoca que se supere el MTU, tendremos un paquete que no cabe en una única trama y se tendrá que fragmentar. La fragmentación debería evitarse cuando sea posible porque aumenta el impacto de los errores de transmisión. Si alguno de los fragmentos se pierde o corrompe, habrá que retransmitir el segmento completo, que de nuevo será fragmentado.

MSS determina el tamaño máximo que tendrá la carga útil de los segmentos durante la conexión. Si por ejemplo, cliente y servidor son vecinos de un mismo enlace Ethernet (que tiene una MTU de 1500 bytes) podrían utilizar un MSS de $1500 - 20 - 20 = 1460$ bytes. Esos 40 bytes que descontamos corresponden a las cabeceras estándar⁴ de TCP e IP.

⁴estándar = sin opciones

Para utilizar un MSS distinto al valor por defecto, un receptor puede incluir la opción TCP MSS (tipo 2) en el primer segmento (el que lleva el flag SYN). Esta opción anuncia cuál es el tamaño máximo de segmento que puede manejar ese extremo. Hay que destacar que esto no es una negociación, no están acordando un valor común para el MSS; cada extremo decide por su cuenta y puede ser un valor diferente en cada uno de los dos sentidos de la comunicación. La opción MSS proporciona un campo de 2 bytes para especificar el tamaño de segmentos, es decir, el valor máximo que se puede anunciar es 64 KiB, aunque el valor 65535 es especial: significa «infinito» y está pensado para utilizar con los *jumbogramas* de IPv6.

En la captura 12.2 puedes ver que los dos mensajes iniciales llevan la opción MSS. El cliente indica 1460 bytes mientras que el servidor indica 1452 bytes.

Como los extremos de una conexión TCP pueden no conocer el MTU de la ruta, podrían elegir valores que obligan a realizar fragmentación. Un router o firewall intermedio podría evitar este problema aplicando una técnica conocida como MSS *clamping*, que consiste en *sobrescribir* el valor de MSS en las cabeceras de la conexión fijando un valor más bajo, adecuado para la ruta que seguirán los paquetes de esa conexión.

El programador también tiene la posibilidad de consultar y modificar el MSS de una conexión establecida. El Listado 12.4 muestra un cliente Python que conecta con un servidor escuchando en el mismo nodo (localhost). Consulta el valor de MSS antes de establecer la conexión y obtiene 536 bytes, que es el valor mínimo. Después, lo vuelve a consultar una vez establecida la conexión y obtiene 32741 bytes. Este valor tan alto se debe precisamente a que ambos cliente y servidor se ejecutan el propio nodo y por eso el SO sabe de antemano que no habrá fragmentación.

```
$ python
>>> import socket
>>> sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
>>> sock.getsockopt(socket.IPPROTO_TCP, socket.TCP_MAXSEG)
536
>>> sock.connect(('127.0.0.1', 2000))
>>> sock.getsockopt(socket.IPPROTO_TCP, socket.TCP_MAXSEG)
32741
```

LISTADO 12.4: TCP: Consulta del MSS de una conexión

Si se fija el valor de MSS antes de establecer la conexión, el socket lo aplicará en la opción MSS de la cabecera al establecerse la conexión. Sin embargo, cambiarlo después normalmente no tiene ningún efecto.

12.3. Desconexión

La desconexión se parece a la conexión en que ambos extremos necesitan asegurarse que el otro ha terminado su tarea y está de acuerdo en dar por terminada la conexión (al menos en la modalidad más *civilizada*). La desconexión sin embargo puede implicar 3 o 4 mensajes, como vamos a ver.

A diferencia de la conexión, que solo la puede iniciar el cliente, la desconexión la puede comenzar cualquiera de los dos extremos en cualquier momento. La desconexión se basa en el envío del flag FIN⁵ que indica que su emisor ha terminado de enviar sus datos, pero a pesar de eso todavía está dispuesto a aceptar nuevos datos. El receptor confirma la recepción de ese mensaje aumentando el número de secuencia en su siguiente confirmación de manera análoga a lo que ocurre con el flag SYN durante la conexión. Si el receptor también ha terminado de enviar datos, ese mismo segmento llevará también el flag FIN. El Listado 12.5 y la Figura 12.2 representan la desconexión con 3 mensajes que acabamos de describir. El Listado 12.6 muestra una captura de una desconexión real.

```
1. --> seq:4000, FIN
2. <-- seq:5000, FIN, ack:4001
3. --> seq:4001, ack:5001
```

LISTADO 12.5: TCP: Secuencia de una desconexión en 3 segmentos

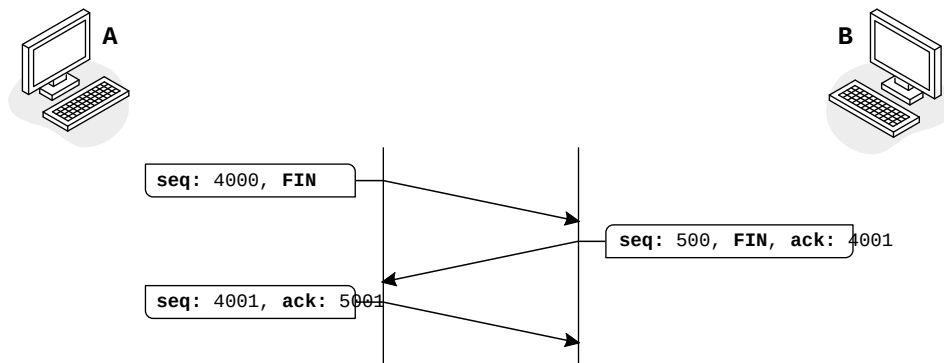


FIGURA 12.2: TCP: Patrón de mensajes de la desconexión en 3 segmentos

⁵por *finish*

```
$ tshark -f "tcp port 80"
[...]
4 192.168.0.37 → 93.184.215.14 TCP 66 50192 → 80 [FIN, ACK] Seq=1 Ack=1 Win=32128 Len=0
5 93.184.215.14 → 192.168.0.37 TCP 66 80 → 50192 [FIN, ACK] Seq=1 Ack=2 Win=65536 Len=0
6 192.168.0.37 → 93.184.215.14 TCP 66 50192 → 80 [ACK] Seq=2 Ack=2 Win=32128 Len=0
```

LISTADO 12.6: TCP: Captura de una desconexión

También puede ocurrir que el receptor del mensaje FIN tenga datos por enviar. En ese caso enviará una confirmación, y continuará enviando sus datos pendientes. Cuando termine emitirá por su cuenta un segmento con el flag FIN. En esta situación la desconexión requiere 4 segmentos (2 + 2), en lugar de 3. El Listado 12.7 y la Figura 12.3 representa una situación en que el nodo que no inicia la desconexión todavía tenía 100 bytes por enviar. Fíjate que el mensaje que lleva datos y su correspondiente confirmación no forman parte de la desconexión; Por eso no se contabilizan en el Listado 12.7 y aparecen en gris en la Figura 12.3.

```
1. --> seq:4000, FIN
2. <-- seq:5000, ack:4001
   <-- seq:5000, len:100
   --> seq:4001, ack:5100
3. <-- seq:5100, FIN
4. --> seq:4001, ack:5101
```

LISTADO 12.7: TCP: Secuencia de una desconexión en 4 segmentos

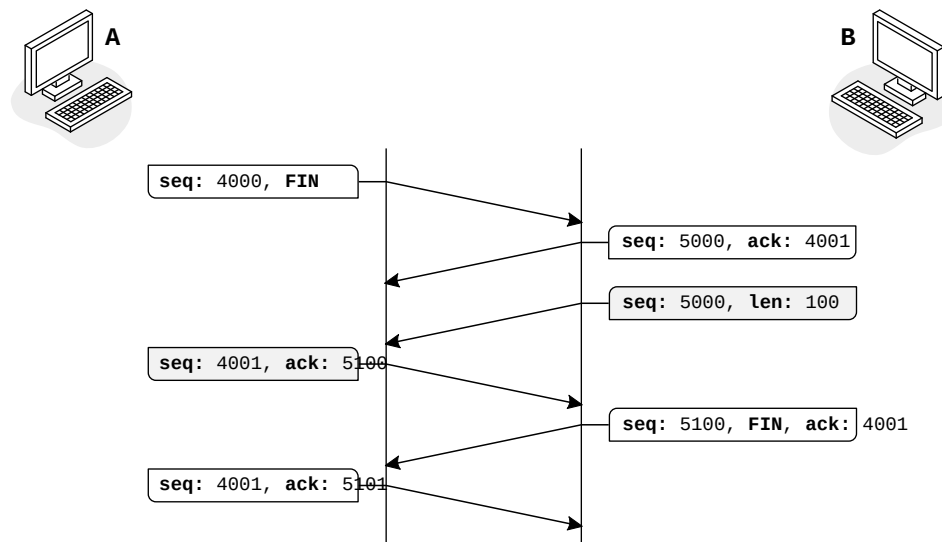


FIGURA 12.3: TCP: Patrón de mensajes de la desconexión en 4 segmentos

Una vez terminada la conexión, los dos extremos pueden liberar sus respectivos TCB, que incluyen los buffers de envío y recepción, y otros recursos asociados.

12.3.1. Tiempo de silencio

Cuando una conexión termina y vuelve a establecerse inmediatamente después, existe una pequeña posibilidad de que los números de secuencia empleados en la nueva conexión se repitan o solapen con los de la conexión anterior. Si esto ocurriera, el receptor podría confundir los segmentos de la nueva conexión con los de la anterior, con resultados impredecibles.

Para evitar este problema, cuando se termina un servidor, el SO impone un *tiempo de silencio* (*quiet time*). Durante ese tiempo el socket se mantiene en un estado de *limbo* (llamado `TIME_WAIT`) que impide vincular un nuevo socket (`bind`) al mismo puerto. Si se intenta levantar de nuevo un servidor en el mismo puerto se obtendrá el mismo error que si realmente estuviera ocupado: *Dirección en uso* (*Address already in use*).

El tiempo de silencio es igual a MSL (Maximum Segment Lifetime), que como su nombre indica, es el tiempo máximo de vida de un segmento, es decir, el tiempo máximo que un segmento puede estar moviéndose por la red antes de llegar a su destino o ser descartado. Se asume que este tiempo es aproximadamente 2 minutos, aunque depende del implementador.

Aunque es una restricción de seguridad importante para un servidor en producción, el programador puede desactivar esta salvaguarda para un socket concreto. Esto es útil por ejemplo para un servidor en su fase de desarrollo o pruebas, y para el que sabe que una confusión entre segmentos no tendrá consecuencias. El Listado 12.8 muestra cómo desactivar el tiempo de silencio.

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

LISTADO 12.8: TCP: Desactivación del tiempo de silencio

Puedes profundizar más en esta cuestión consultado la sección «The TCP Quiet Time Concept» en la RFC 793 [24].

12.4. Reset

La utilidad principal del flag RST es rechazar una conexión entrante. Ante un intento de conexión (segmento con el flag SYN), el SO del nodo receptor responderá con un segmento con el flag RST si la conexión no puede ser establecida. Esto puede ocurrir por varios motivos:

- El puerto destino está cerrado, es decir, no está vinculado a ningún proceso servidor. Puedes ver un ejemplo de este caso en el Listado 12.9.
- Hay un servidor, pero no tiene posibilidad de aceptar nuevas conexiones en ese momento, quizá por una limitación de recursos.
- El segmento de inicio de conexión no es válido.

```
$ tshark -f "tcp por 2000"
1 192.168.1.235 → 192.168.1.1 TCP 74 48318 → 2000 [SYN] Seq=0 Win=32120 Len=0 MSS=1460
2 192.168.1.1 → 192.168.1.235 TCP 60 2000 → 48318 [RST, ACK] Seq=1 Ack=1 Win=0 Len=0
```

LISTADO 12.9: TCP: Rechazo de un intento de conexión a un puerto cerrado

El flag RST también se utiliza para cerrar una conexión activa de forma inmediata sin esperar a que el otro extremo esté de acuerdo, de hecho ni siquiera espera una confirmación. Esto se puede hacer en situaciones anómalas, cuando el otro extremo está infringiendo las reglas del protocolo, como por ejemplo usando números de secuencia incorrectos, flags ausentes o inesperados, etc. Todas estas situaciones las maneja automáticamente el SO y no es necesario que el programador intervenga.

Pero el programador también puede usar RST para finalizar la conexión de manera abrupta aunque esté funcionando correctamente. Obviamente puede conllevar la pérdida de datos si el otro extremo no ha terminado de enviar, pero es un riesgo que el emisor asume. El Listado 12.10 muestra cómo desactivar la «permanencia» (*linger*) de la conexión hasta recibir todos los datos. De este modo, el método `close()` cierra la conexión con un segmento RST en lugar del método convencional que hemos visto en §12.3.

```
import socket, struct

sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
sock.setsockopt(socket.SOL_SOCKET, socket.SO_LINGER, struct.pack('ii', 1, 0))
sock.connect((host, port))
[... envío de datos ...]
sock.close() # envía RST, no habrá handshake de desconexión
```

LISTADO 12.10: TCP: Cerrar una conexión de forma abrupta con RST

El SO del otro extremo (el que recibe el RSTT) enviará al proceso un error de tipo `errno.ECONNRESET` para indicarle esta situación.

12.5. Control de flujo en TCP

El control de flujo en TCP se basa en el mecanismo de *ventana deslizante* similar al que hemos visto antes en este mismo capítulo. El receptor se preocupa de informar en cada mensaje del espacio disponible (en bytes) es su buffer de recepción —a esto se le llama «actualizar la ventana». Esa cantidad de espacio libre es lo que llamamos *ventana de recepción* y es el valor que aparece en el campo *window* de la cabecera TCP. Determina también los números de secuencia de los bytes que puede recibir. Por ejemplo, si el campo *window* tiene un valor 1000 y el campo *acknowledge number* tiene un valor de 3000, significa que el receptor puede aceptar los bytes desde el 3000 hasta el 3999, y no otros. Al recibir este mensaje, el emisor sabe que como mucho podrá enviar segmentos por un total de esos 1000 bytes a partir del último byte confirmado. De este modo, el receptor TCP influye continuamente en la tasa de salida del emisor. A este mecanismo concreto es a lo que llamamos *control de flujo de receptor*, aunque en este capítulo lo llamaremos simplemente *control de flujo*.

La Figura 12.4 representa el flujo de datos completo desde que salen del proceso emisor hasta que son consumidos por el proceso receptor. Recuerda que estamos representando solo un sentido de la comunicación de una única conexión. Cada conexión establecida conlleva una estructura como esta en cada sentido de la comunicación.

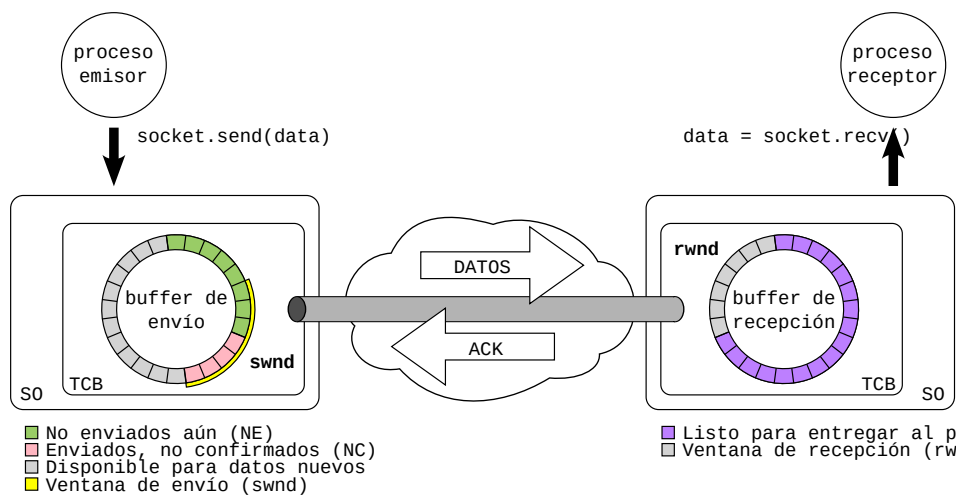


FIGURA 12.4: TCP: Flujo de datos
(se representa un solo sentido de la comunicación)

El emisor mantiene el buffer de envío (*sending buffer*) implementado como una cola circular. Los datos entran en este buffer vía el método `socket.send()` y salen cuando el SO decide construir un nuevo segmento y enviarlo a la red, pero no hay una relación directa entre la invocación de `send()` y el envío de segmentos, y de hecho varios `send()` podrían corresponder a uno, varios o ningún segmento en ese instante.

El buffer de envío está formado por:

- Espacio disponible para almacenar datos nuevos procedentes del proceso emisor.
- Bytes enviados que aún no han sido confirmados. Nos referiremos a esta parte del buffer como NC (no confirmados).
- Bytes que aún no han sido enviados. Nos referiremos a esta parte del buffer como NE (no enviados).

El buffer de recepción solo tiene datos en dos estados: recibidos confirmados y espacio libre. Cuando el computador recibe mensajes a través de una interfaz de red, el SO procesa el segmento, identifica el proceso al que corresponde el puerto destino que aparece en la cabecera de transporte y finalmente inserta su carga útil en este buffer. Los datos salen del buffer cuando el proceso receptor los solicita mediante el método `socket.recv()`.

Eso significa que si un proceso receptor, que mantiene una conexión abierta, deja de recoger datos —con `recv()`— mientras el emisor sigue enviando, ambos buffers (envío y recepción) se llenarán y como consecuencia el emisor quedará bloqueado en una invocación a `send()`.

Es importante aclarar que el SO envía el mensaje de confirmación cuando los datos son almacenados en el buffer de recepción, es decir, antes de que hayan sido procesados por el proceso receptor. Si el proceso receptor no llega a pedirlos y termina, el buffer de recepción se destruye y los datos se pierden. Dicho de otro modo, el mecanismo garantiza que los datos llegan al SO destino, no al proceso receptor.

El tamaño de estos buffers depende de cada SO y de su configuración específica. Puedes consultar estos valores para un socket concreto por medio de los identificadores `socket.SO_SNDBUF` y `socket.SO_RCVBUF`. El Listado 12.11 muestra una *shell* Python con el código que permite conseguirlo. Los valores mostrados: 16 384 y 131 072 bytes, son los valores reales (por defecto) que se han obtenido en un sistema Debian con Linux 6.12.9.

El programador puede modificar el tamaño de estos buffers, aunque si lo pruebas (como en el listado) verás algo curioso. El SO reserva el doble de

lo que pides. Esto se debe a que debe contar con el espacio necesario para almacenar cabeceras y metadatos, que estima que puede llegar a ser el doble de la carga útil. Estos tamaños tienen un valor mínimo y máximo que depende del SO. En Linux el mínimo ronda los 2-4 kB y un máximo 400 kB, y estas primitivas truncarán valores fuera de ese rango.

También es posible consultar el tamaño de los buffers, y muchos otros datos, para un socket activo con el comando `ss` como veremos un poco más adelante.

```
$ python
>>> import socket
>>> sock = socket.socket()
>>> sock.getsockopt(socket.SOL_SOCKET, socket.SO_SNDBUF)
16384
>>> sock.getsockopt(socket.SOL_SOCKET, socket.SO_RCVBUF)
131072
>>> sock.setsockopt(socket.SOL_SOCKET, socket.SO_SNDBUF, 4096)
>>> sock.setsockopt(socket.SOL_SOCKET, socket.SO_RCVBUF, 4096)
>>> sock.getsockopt(socket.SOL_SOCKET, socket.SO_SNDBUF)
8192
>>> sock.getsockopt(socket.SOL_SOCKET, socket.SO_RCVBUF)
8192
```

LISTADO 12.11: TCP: Consultar y modificar el tamaño de los buffers

12.6. Ventanas de envío y recepción

Acabamos de ver que la *ventana de recepción* (abreviado como *rwnd*) es el espacio libre (no ocupado) en el buffer de recepción. De la otra parte, la *ventana de envío* (abreviado como *swnd*) define la **parte concreta** del buffer de envío (los bytes específicos) que el emisor tiene permitido enviar en ese momento. El tamaño de la ventana de envío está limitado en todo momento por la ventana de recepción.

$$\text{swnd} \leq \text{rwnd} \quad (12.1)$$

Como la velocidad a la que el proceso receptor consume datos del buffer de recepción puede variar drásticamente a lo largo del tiempo, la ventana de recepción puede cambiar del mismo modo, y a su vez estos cambios, notificados al emisor, condicionan la ventana de envío. Dicho de otro modo, la tasa a la que el proceso receptor consume datos condiciona indirectamente la tasa a la que el proceso emisor puede enviar. La ventana de envío por tanto puede crecer, decrecer o incluso cerrarse en cualquier momento.

Control de flujo en una conexión real

Para ver el control de flujo en acción vamos a usar el ejemplo `flow-control`. Consta de un servidor que únicamente recibe, pero que nos da la opción de limitar la tasa a la que consume. El siguiente comando arranca el servidor en el puerto 2000 y limita la tasa a 200 kB/s.

```
code/flow-control$ ./server.py --limit 200 2000
```

Y un cliente que se conecta al servidor indicado y sólo envía. Lo hace tan rápido como pueda. Para ejecutarlo, abre otra terminal y hazlo así:

```
code/flow-control$ ./client.py 127.0.0.1 2000
```

El cliente informa del tamaño del buffer de envío, la cantidad de datos que ha enviado (pero que pueden estar aún en el buffer) y la tasa media de envío. El servidor informa del tamaño del buffer de recepción, la cantidad de datos recibidos y la tasa media de recepción. Lo puedes ver en acción en la Figura 12.5. Más adelante en el capítulo 16 veremos cómo miden y limitan la tasa, de momento solo los vamos a usar para generar tráfico.

Cliente

```
~$ ./client.py 127.0.0.1 2000
Sending buffer size: 2,626 kB
(-) sent:10,630 kB, rate:268.2 kB/s
```

Servidor

```
~$ ./server.py 2000 200
Receiving buffer size: 131 kB
received:9,067 kB, rate:200.1 kB/s
```

FIGURA 12.5: Flujo continuo con tasa limitada por el receptor (servidor) — `flow-control`

Si los ejecutas ambos en tu computador, tanto el emisor como «la red» pueden enviar más rápido de esa tasa de 200 kB/s. Por eso, tanto el buffer de recepción del servidor como el de envío del cliente eventualmente se llenan, lo que produce un bloqueo intermitente del emisor durante unos segundos. Cada bloqueo (cuando se detiene la cuenta de bytes enviados) corresponde al momento en que el receptor cierra la ventana. Esto es una prueba clara de cómo el receptor condiciona absolutamente la capacidad del emisor para enviar datos, a punto de detenerlo por completo.

Una vez tienes en marcha los dos programas en sendas consolas, puedes usar el comando `ss` que consulta el TCB del socket para obtener muchos datos interesantes. El comando concreto que se muestra filtra la conexión establecida que incluye el socket cliente y el socket conectado en el lado del cliente, es decir, los flujos con destino y origen en el puerto 2000. Si no se aplica un filtro, listaría todos los sockets activos.

```
$ ss -tin 'sport = :2000 or dport = :2000'
```

State	Recv-Q	Send-Q	Local Address:Port	Peer Address:Port
ESTAB	0	2260096	127.0.0.1:33188	127.0.0.1:2000
wscale:7,7 rto:220 backoff:1 rtt:17.291/0.018 mss:54976 bytes_sent:58781120 bytes_retrans:164928 bytes_acked:58616193 segs_out:1608 segs_in:1602 data_segs_out:1073 send 254356602bps lastsnd:512 lastrcv:293028 lastack:272 pacing_rate 508687456bps delivery_rate 50765624bps delivered:1074 busy:293024ms rwnd_limited:293016ms(100.0%) retrans:0/3 dsack_dups:3 rcv_space:65495 rcv_ssthresh:65495 notsent:2260096 minrtt:0.01 rcv_wnd:65536				
ESTAB	7552	0	127.0.0.1:2000	127.0.0.1:33188
wscale:7,7 rto:200 rtt:0.017/0.008 ato:40 mss:32768 bytes_received:58616192 segs_out:1601 segs_in:1608 data_segs_in:1073 send 154202352941bps lastsnd:293028 lastrcv:512 lastack:512 pacing_rate 308404705880bps delivered:1 app_limited rcv_rtt:1 rcv_space:65483 rcv_ssthresh:109848 minrtt:0.017 snd_wnd:65536				

LISTADO 12.12: TCP: Información de la conexión con ss

En la primera línea aparece la cabecera con unos campos que se aplican a todos los sockets listados. Si te fijas solo en esas filas puedes observar lo siguiente:

State	Recv-Q	Send-Q	Local Address:Port	Peer Address:Port
ESTAB	0	2260096	127.0.0.1:33188	127.0.0.1:2000
ESTAB	7552	0	127.0.0.1:2000	127.0.0.1:33188

Estas columnas indican:

- State: es el estado de la conexión. En ambos es ESTAB, es decir, la conexión está establecida.
- Recv-Q: es la ocupación (en bytes) de la cola de recepción. La primera fila corresponde con el cliente, que no recibe nada, y la segunda con el servidor.
- Send-Q es la ocupación de cola de envío.
- Local Address:Port es la dirección IP y el puerto local del socket.
- Peer Address:Port es la dirección IP y el puerto del socket remoto.

La lista de variables que hay debajo de esas filas corresponden a cada socket. Incluimos solo la que resultan relevantes en este momento. La Tabla 12.1 muestra los datos del cliente mientras que la Tabla 12.2 muestra los del servidor.

En próximas secciones y capítulos veremos otros datos interesantes que aparecen es la información que ofrece ss.

12.7. RTO: el temporizador de retransmisión

En un protocolo confiable de la capa de enlace, la duración del temporizador de retransmisión es fácil de calcular. Como el envío de datos se realiza directamente a un vecino, resulta sencillo determinar el tiempo necesario para que un mensaje de datos llegue al destino y, de vuelta, la confirmación

Variable	Valor	Descripción
mss	54 976	Tamaño máximo de segmento (MSS)
bytes_sent	58 781 120	Bytes enviados
bytes_retrans	164 928	Bytes retransmitidos
bytes_acked	58 616 193	Bytes confirmados (ACK)
segs_out	1 608	Segmentos totales
segs_in	1 602	Segmentos totales
data_segs_out	1 073	Segmentos de datos
send	25 435	Tasa de envío (Mbps)
delivery_rate	5 076	Tasa de entrega (Mbps)
delivered	1 074	Segmentos entregados
rwnd_limited	293 016	La tasa está limitada por <code>rwnd</code> (ms) – 100 %
rcv_space	65 495	Espacio en el búfer de recepción
rcv_wnd	65 536	Tamaño de la ventana de recepción

CUADRO 12.1: Métricas del estado del socket cliente relacionadas con buffers y ventanas — `🔗/flow-control`

Variable	Valor	Descripción
mss	32 768	Tamaño máximo de segmento (MSS)
bytes_received	58 616 192	Total de bytes recibidos
segs_out	1 601	Segmentos totales enviados
segs_in	1 608	Segmentos totales recibidos
data_segs_in	1 073	Segmentos de datos recibidos
send	154.20	Tasa de envío (Gbps)
delivered	1	Segmentos entregados
app_limited	-	Indica limitación por la aplicación
snd_wnd	65 536	Tamaño de la ventana de envío

CUADRO 12.2: Métricas de estado del socket servidor relacionadas con buffers y ventanas — `🔗/flow-control`

correspondiente. Considerando el tamaño del mensaje y el ancho de banda del enlace se puede calcular este tiempo con precisión. Se debe añadir obviamente el tiempo de procesamiento de los mensajes en los extremos, su almacenamiento, etc., pero en general todo esto es bastante predecible, determinista y constante.

Con TCP la situación es muy distinta [25]. El envío de un segmento a un destino arbitrario depende de muchas variables ya que el segmento debe atravesar probablemente muchas redes y dispositivos intermedios:

- La distancia física entre emisor y receptor.
- La topología de la red y la ruta que van a tomar los datagramas.
- La carga de la red.
- El ancho de banda de los enlaces.
- La posible congestión en la red.
- La carga de trabajo de los routers intermedios, y de los nodos finales.

Y la situación es más compleja aún. Todas estas variables toman valores diferentes en función del destino y ruta, y pueden cambiar rápida y significativamente a lo largo de una misma conexión por cambios en las condiciones en distintos lugares de la interred.

Para resolverlo, TCP mide constantemente el RTT, es decir, el tiempo que transcurre desde que se envía un segmento hasta que llega su correspondiente confirmación. A partir de esta medida, calcula el valor del temporizador de retransmisión (RTO) que va a aplicar. Así pues, es posible que se el período de retransmisión sea distinto incluso para segmentos consecutivos. Para la obtención del valor final de RTO, TCP calcula una versión ‘suavizada’ (SRTT) del RTT con la fórmula 12.2. El suavizado proporciona estabilidad al cálculo del RTO, evitando que variaciones puntuales en la medida provoquen cambios bruscos.

$$SRTT = (1 - \alpha) \cdot SRTT + \alpha \cdot RTT \quad (12.2)$$

El coeficiente α es el *factor de suavizado*. Un valor entre 0 y 1 que determina si se le da más importancia al valor RTT que se acaba de medir (α alto) o si por el contrario es más importante el valor SRTT previo (α bajo). El valor de α suele ser 1/8, es decir, conservador. Para el cálculo inicial $SRTT = RTT$.

TCP también calcula RTTVAR, que representa la desviación media del RTT respecto a SRTT. Permite estimar la variabilidad de RTT según la fórmula 12.3 (algoritmo de Jacobson [26]).

$$RTTVAR = (1 - \beta) \cdot RTTVAR + \beta \cdot |SRTT - RTT| \quad (12.3)$$

La fórmula es similar a la de SRTT, β también un factor de suavizado cuyo valor habitual es 1/4. Para el cálculo inicial se toma $RTTVAR = RTT/2$. Esta variable se actualiza antes que SRTT al llegar un ACK adecuado.

A partir de SRTT y RTTVAR se calcula el RTO según la fórmula 12.4.

$$RTO = SRTT + \max(G, K \cdot RTTVAR) \quad (12.4)$$

donde:

G es la precisión del reloj (*p. ej.* 1ms).

K es un factor de ponderación, normalmente 4.

Lo que dice esta fórmula es que el RTO es el SRTT más un margen de seguridad que será mayor cuanto más variable sea la latencia de la red (mayor sea RTTVAR). Adicionalmente se aplica un valor mínimo de 1 segundo y un valor máximo de 60 segundos.

TCP utiliza el algoritmo de Karn [27], que estipula que no se deben realizar medidas de RTT con segmentos retransmitidos, ya que la ambigüedad entre posibles ACK correspondientes a segmentos duplicados podría falsear el cálculo. En todo caso, no se recalcula el RTO para cada segmento, se realiza solo una vez por cada ronda. La única excepción a esta norma es la opción TCP Timestamp.

Cada emisor TCP mantiene un único RTO, que está asociado al primer segmento enviado no confirmado (SND.UNA). Eso implica que si todos los datos están confirmados, el RTO se desactiva. Cuando se envía un nuevo segmento, se reactiva el temporizador con el último valor RTO calculado. Cuando llega un ACK, se reinicia el temporizador con el valor del RTO actualizado.

Cuando el RTO expira, el emisor retransmite el segmento no confirmado más antiguo y duplica el valor del RTO. Si vuelve a expirar se duplicará de nuevo hasta un máximo según la fórmula 12.5. Esto se denomina *backoff exponencial* y pretende adaptar las retransmisiones a las condiciones de la red.

SND.UNA

Es el acrónimo de Send Unacknowledged Number, y es el número de secuencia relacionado con el segmento más antiguo enviado y no confirmado. Más concretamente, indica el límite inferior de la ventana de envío.

$$RTO = \min(2 \cdot RTO, RTO_{max}) \quad (12.5)$$

donde:

RTO_{max} es el valor máximo que puede tomar el RTO, normalmente 60s.

Cuando el emisor consiga realizar una nueva medida de RTT, es decir, cuando pueda enviar un nuevo segmento (no una retransmisión) y éste sea confirmado, el valor de RTO volverá a sus cotas habituales.

Ejemplo de cálculo de RTO

Veamos un pequeño ejemplo que ilustra cómo evoluciona el valor del RTO. Supongamos un valor calculado de $SRTT = 100ms$, $RTTVAR = 25ms$ que resulta en $RTO = SRTT + 4 \cdot RTTVAR = 200ms$. Cuando correspond, se recalculan SRTT, RTTVAR y RTO según las fórmulas 12.2, 12.3 y 12.4.

- Se envían 3 segmentos S1 en $t = 0ms$, S2 en $t = 2ms$ y S3 en $t = 2ms$, asumiendo que la ventana así lo permite. El RTO queda asociado a S1.
- En $t = 90$ llega el ACK de S1. El RTT medido es 90ms. Se recalcula RTO para S2 en 183,75ms, expira en el instante $t = 273,75$.
- En $t = 120$ lleva un ACK duplicado para S1. No tiene ningún efecto.
- En $t = 273,75$ expira el temporizador de S2. Se aplica *backoff* y RTO se duplica hasta 367,5ms. Expirará en el instante $t = 641,25$.
- En $t = 330$ llega un ACK para S3. No se recalcula el RTO porque según Karn, el RTT medido debe descartarse. Como los tres segmentos enviados están confirmados, el RTO se desactiva.
- En $t = 400$ se envía S4. El temporizador se reinicia con el valor del RTO actualizado (367,5mstt). Expirará en $t = 767,5$.
- En $t = 500$ llega el ACK de S4. Se recalcula el valor RTO con una nueva muestra válida, resultando en 178ms, y se desactiva porque no hay confirmaciones pendientes.

La Tabla 12.3 resume el ejemplo y la evolución de las variables implicadas. Todos los valores están expresados en ms.

t	evento	RTT	RTTVAR	SRTT	RTO	expira en
0,00	envío S1	—	25,00	100,00	200,00	200,00
1,00	envío S2	—	25,00	100,00	—	—
2,00	envío S3	—	25,00	100,00	—	—
90,00	ACK S1	90	21,25	98,75	183,75	273,75
120,00	ACK (dup) S1	—	21,25	98,75	183,75	273,75
273,75	timeout S2	—	21,25	98,75	367,50	641,25
330,00	ACK (acum) S3	—	21,25	98,75	367,50	—
400,00	envío S4	—	21,25	98,75	367,50	767,50
500,00	ACK S4	100	16,25	98,91	163,91	—

CUADRO 12.3: TCP: Evolución de las variables en el ejemplo de RTO

RTO en una conexión real

Con `ss` se puede consultar información importante sobre el cálculo de RTO. Retomemos el ejemplo de la sección 12.6 y veamos las variables del cliente relacionadas en la Tabla 12.4.

Variable	Valor	Descripción
rto	220	Valor actual del RTO (ms)
backoff	1	Multiplicador de tiempo de espera (por timeout)
rtt	17.291 / 0.018	RTT medido (promedio / desviación) ms
minrtt	0.01	RTT mínimo observado

CUADRO 12.4: Métricas del estado del socket cliente relacionadas con el RTO—

🔗/flow-control

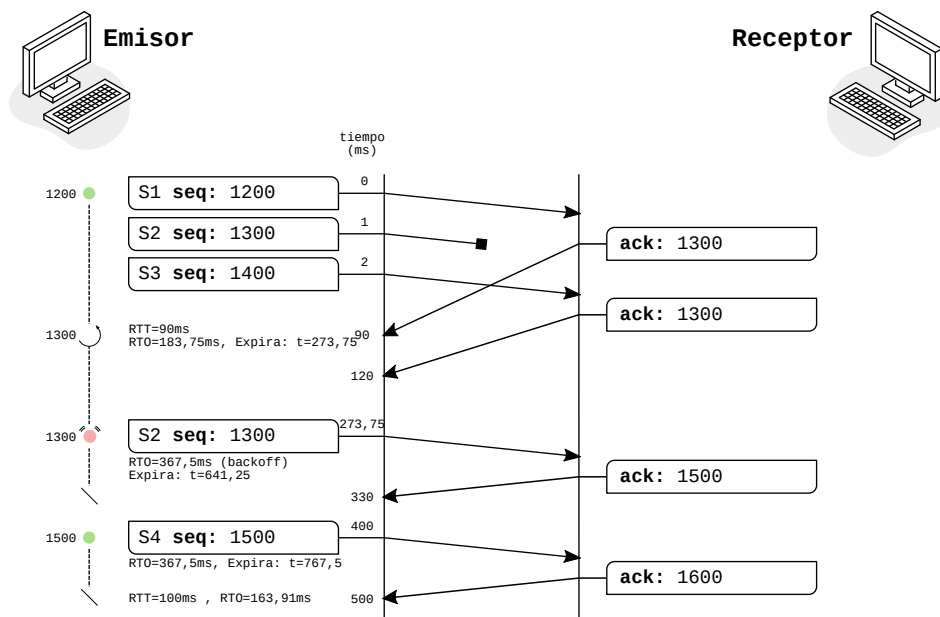


FIGURA 12.6: TCP: Ejemplo de cálculo del RTO

12.8. El síndrome de la *ventana tonta*

El *síndrome de la ventana tonta* (o SWS) es un efecto que conlleva una importante ineficiencia en la transferencia de datos propiciada por el envío de segmentos cada vez más pequeños. Ambos, receptor y emisor, son responsables de este efecto: el receptor porque anuncia tamaños de ventana pequeños tan pronto como dispone de algo de espacio en el buffer de recepción, y el emisor porque envía segmentos aunque tenga pocos datos disponibles. Por eso, resolver el problema requiere la colaboración de ambos extremos.

El receptor puede ayudar a paliar el problema esperando a disponer de una cantidad de espacio libre significativa antes de anunciarlo (normalmente entre MSS y la mitad del buffer de recepción) y reservando espacio libre adicional que no se anuncia. Además el receptor puede hacer uso del ACK retardado (con un máximo de medio segundo) y así favorecer la liberación de espacio adicional.

Por su parte, el emisor puede esperar a disponer de una cantidad significativa de datos antes de enviar un nuevo segmento. El algoritmo de Nagle [28] se ocupa de esto. Lo que dice Nagle básicamente es:

«Si el buffer de envío ya contiene datos sin confirmar, los nuevos datos que lleguen desde el proceso emisor se añaden al buffer hasta que se confirmen los datos pendientes o bien se acumulen MSS bytes nuevos».

12.9. Cierre de la ventana

El cierre de la ventana de recepción ocurre cuando el receptor anuncia un valor de 0 bytes en el campo `window` de la cabecera TCP, y provoca una situación peculiar. Esta acción suspende el tráfico (en ese sentido de los datos) a pesar de que el emisor tenga aún datos que enviar. Termina cuando el receptor, en algún momento, envía una confirmación indicando un valor distinto de 0. Pero hay un problema: ¿qué ocurre si se pierde esa confirmación? El emisor quedará esperando indefinidamente el mensaje de apertura, que no llegará; y el receptor queda esperando que el emisor envíe datos nuevos, que tampoco van a llegar, es decir, tenemos un bloqueo.

Para evitarlo, TCP dispone de un mecanismo conocido como *prueba de ventana cero*. Consiste en el uso del llamado *temporizador de persistencia*, que al expirar, le dice al emisor que envíe una *prueba de ventana cero* (Zero Window Probe, ZWP), que puede ser un segmento sin datos o una retransmisión sobre datos ya confirmados. Eso fuerza al receptor a enviar una confirmación que permite al emisor saber si la ventana sigue cerrada. Emisor y receptor deberían permitir que la ventana quede cerrada indefinidamente. La figura 12.7 muestra un ejemplo de este mecanismo.

El temporizador de persistencia comienza siendo igual a RTO, pero como el RTO, se duplica cada vez que expira, hasta un valor máximo de 60 segundos.

12.10. Confirmación selectiva

Con la confirmación convencional que hemos visto, cuando se pierde un segmento, pero los siguientes llegan correctamente, el emisor recibe repetidamente confirmaciones para el último segmento que llegó correcto y en orden (confirmaciones duplicadas), y conforme vaya expirando el temporizador para los segmentos afectados, los irá retransmitiendo, a pesar de que hubieran llegado correctamente (el emisor no puede saberlo), y seguirá haciéndolo hasta recibir una confirmación sobre datos nuevos. Eso por supuesto conlleva retransmisiones innecesarias y un desperdicio significativo de tiempo y recursos.

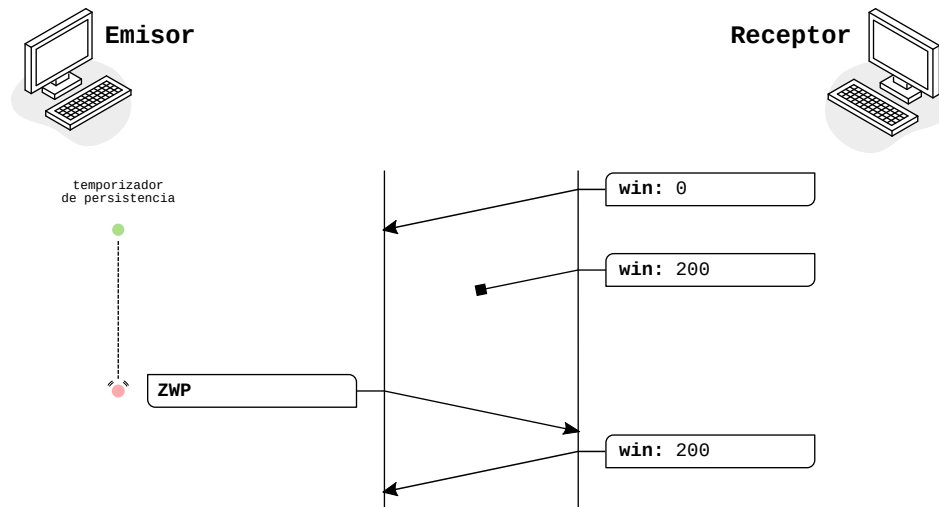


FIGURA 12.7: TCP: «Prueba de ventana cero» tras expirar el temporizador de persistencia

Para mejorar esta situación, el receptor puede enviar *confirmaciones selectivas*⁶ o SACK. En ese caso, el receptor puede informar de los segmentos que han llegado correctamente incluso fuera de orden. Entonces el emisor puede enviar inmediatamente los segmentos que faltan sin tener que esperar a que expire el temporizador.

Para proporcionar esta funcionalidad, TCP dispone de dos opciones. La primera se llama SACK-permitted (tipo 4) y se incluye en el segmento SYN para indicar que quién la emite es capaz de procesar confirmaciones selectivas. La segunda opción es la confirmación SACK en sí misma (tipo 5), que lógicamente solo se debería enviar a un emisor que haya indicado que puede manejarlas.

La opción SACK contiene una lista de bloques que han llegado correctamente. Cada bloque (que puede incluir varios segmentos) se especifica con los números de secuencia del primer byte del bloque y del siguiente al último. Como cada número de secuencia requiere 4 bytes, la opción ACK puede contener un máximo de 4 bloques. Cuando el emisor retransmite los segmentos que faltan, envían una confirmación convencional actualizada.

La Figura 12.8 muestra un ejemplo de confirmación SACK en la que el receptor informa que ha recibido correctamente los datos fuera de orden (con números de secuencia 1601 a 2000). En emisor procede a retransmitir

⁶No confundir con *repetición selectiva*

inmediatamente el segmento que falta (con número de secuencia 1401) sin esperar a que expire el temporizador.

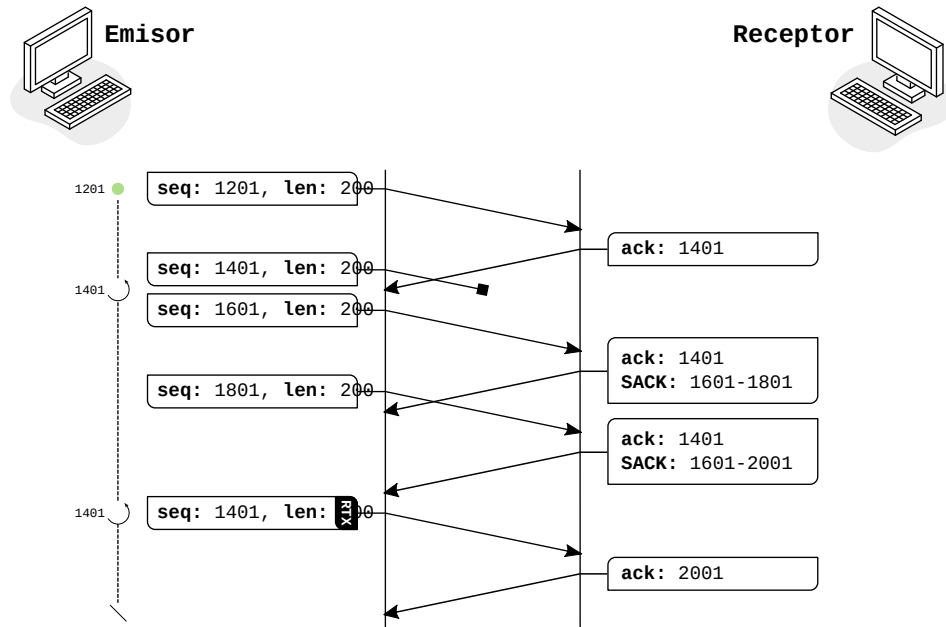


FIGURA 12.8: TCP: Confirmación selectiva

12.11. Aplicaciones interactivas

TCP transmite un flujo continuo de bytes entre emisor y receptor. El emisor (gracias al algoritmo de Nagle entre otros) trata de realizar una gestión óptima que maximice el uso del canal respetando, como hemos visto, las limitaciones del receptor y evitando efectos negativos como el envío de segmentos demasiado pequeños. El SO decide cuando enviar cada segmento y qué cantidad de bytes colocar en cada uno.

Sin embargo, esta búsqueda de la eficiencia es muy perjudicial para las aplicaciones interactivas tales como shell remota, escritorio remoto o videojuegos. Si por ejemplo movemos el ratón o pulsamos una tecla y esperamos ver el resultado de forma inmediata, el segmento que lleva esos datos al servidor debe salir inmediatamente, no puede esperar a que se acumulen MSS bytes. Este tipo de aplicaciones serían inviables.

Para lograrlo, hay varias cosas que podemos hacer. Una de ellas es desactivar el algoritmo de Nagle para la conexión particular que lo requiera. Puedes ver cómo hacerlo en Python en el Listado 12.13.

```
sock.setsockopt(socket.IPPROTO_TCP, socket.TCP_NODELAY, 1)
```

LISTADO 12.13: TCP: Desactivación del algoritmo de Nagle

Además de desactivar Nagle, TCP proporciona en su cabecera 2 flags que modifican el comportamiento habitual y que pueden ayudar a llevar los datos hasta el receptor lo antes posible.

Cuando el emisor incluye el flag PSH (*push*) en el segmento, el receptor lo interpreta como una señal de que debe entregar los datos a la capa superior inmediatamente, sin esperar a nuevos mensajes, pero no tiene ningún efecto en el emisor, es decir, no le obliga a enviar el segmento antes de lo habitual. El flag PSH puede ser añadido automáticamente por el SO en ciertas circunstancias, por ejemplo, si los datos llevan determinado tiempo en el buffer de envío.

El flag URG (*urgent*) indica que la primera parte de la carga útil del segmento deben tratarse de forma prioritaria. El flag habilita el campo *puntero urgente* que corresponde con el offset (sumado al número de secuencia) que indica dónde acaban los datos urgentes (es el puntero al último byte urgente). En todo caso, es un mecanismo que históricamente ha tenido interpretaciones diferentes en las distintas implementaciones, ha tenido poco uso y no se recomienda.

12.12. Control de errores

En esta sección veremos cómo TCP resuelve los errores que pueden surgir durante una conexión.

12.12.1. Segmento perdido/corrupto

A efectos prácticos no hay diferencias entre un segmento que llega a su destino con un checksum incorrecto (corrupto) y uno que nunca llega. El receptor no puede procesar la información que contiene un segmento corrupto y por tanto lo descarta, de modo que las consecuencias son las mismas. En ambos casos, el receptor no envía confirmación, el RTO expirará y el emisor lo retransmitirá. La Figura 12.9 ilustra esta situación.

12.12.2. Confirmación perdida/corrupta

Una confirmación corrupta es descartada igual que cualquier otro segmento corrupto. La consecuencia, como en el caso anterior, es que el RTO expirará y habrá una retransmisión. Sin embargo, como las confirmaciones son

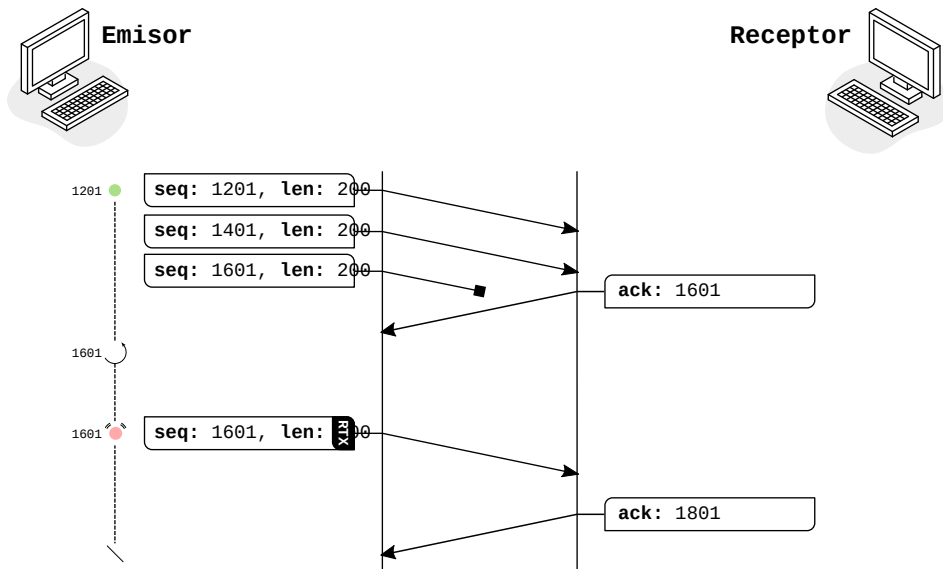


FIGURA 12.9: TCP: Segmento perdido

acumulativas, si llegara una confirmación sobre un número de secuencia posterior, el segmento quedaría confirmado a pesar de todo y la pérdida/corrupción del segmento de confirmación no tendría ninguna consecuencia. Puedes ver un ejemplo de esto en la Figura 12.10

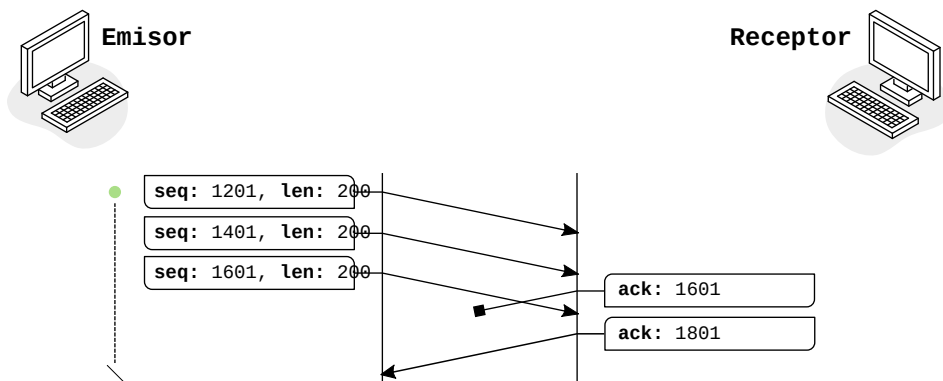


FIGURA 12.10: TCP: Confirmación perdida

12.12.3. Segmento duplicado

Cuando una confirmación se pierde y no existe otra que la sustituya de forma acumulativa, el emisor retransmite el segmento correspondiente y

el receptor recibirá un duplicado. En esta situación el receptor ignora el duplicado, pero está obligado a enviar una confirmación inmediatamente (ver Figura 12.11).

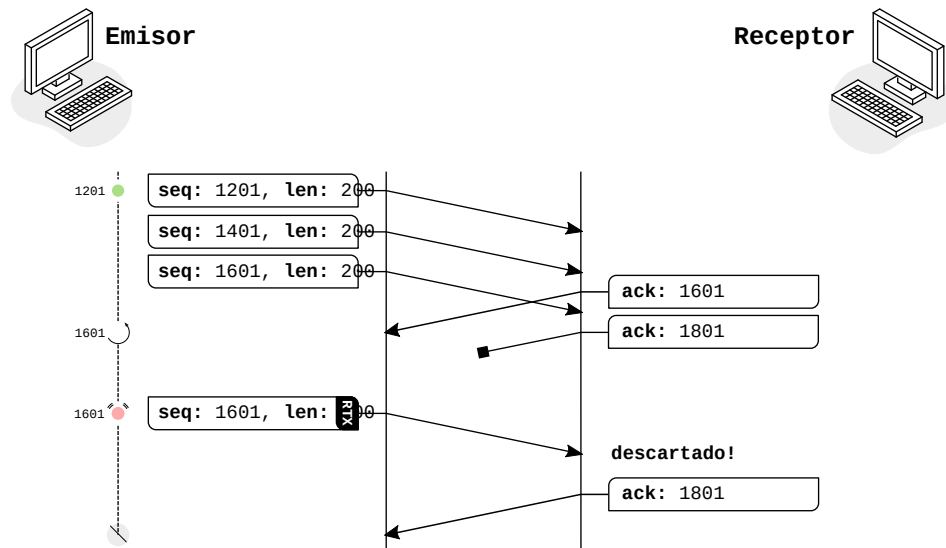


FIGURA 12.11: TCP: Segmento duplicado

12.12.4. Segmento fuera de orden

En una red de conmutación de paquetes como Internet es posible que segmentos de un mismo flujo o conexión lleguen al receptor en un orden diferente al que fueron enviados (Figura 12.12).

Esto puede ocurrir por varias razones, por ejemplo, porque el segmento que llega tarde se perdió y fue retransmitido, o porque el segmento que llegó antes se retrasó por alguna razón (congestión, colas en los routers, etc.). En cualquier caso, el receptor no puede procesar un segmento fuera de orden hasta que lleguen los segmentos anteriores.

Como hemos visto anteriormente, el receptor almacenará los segmentos fuera de secuencia en el buffer de recepción y enviará confirmaciones para el último segmento que llegó respetando la secuencia. Cuando finalmente llegue el segmento que falta, el receptor enviará una confirmación por todo el conjunto.

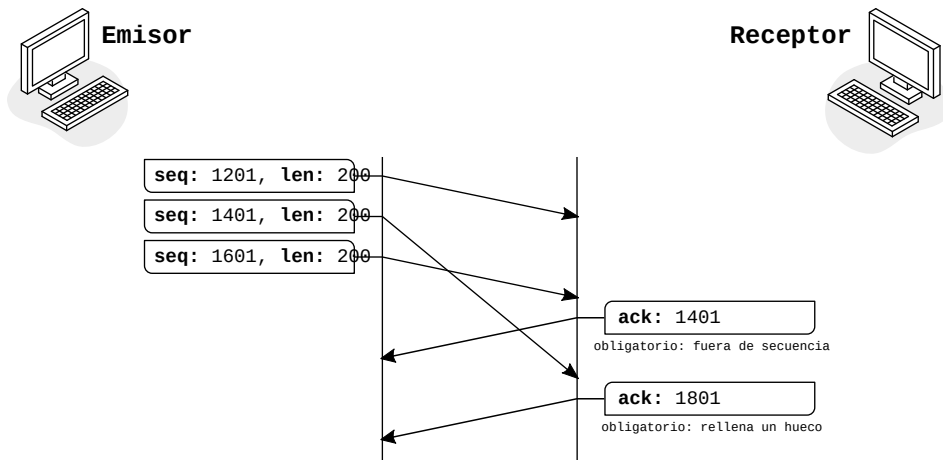


FIGURA 12.12: TCP: segmento fuera de orden

12.12.5. Confirmación duplicada

Tanto si un segmento de datos se pierde como si llega fuera de orden, el receptor enviará varias veces la misma confirmación para los siguientes segmentos que lleguen correctamente a continuación. Al emisor no le afectan las confirmaciones duplicadas, pero como veremos más adelante, le dan información valiosa sobre el estado de la red. Puedes ver un escenario en el que aparecen confirmaciones duplicadas en la Figura 12.13

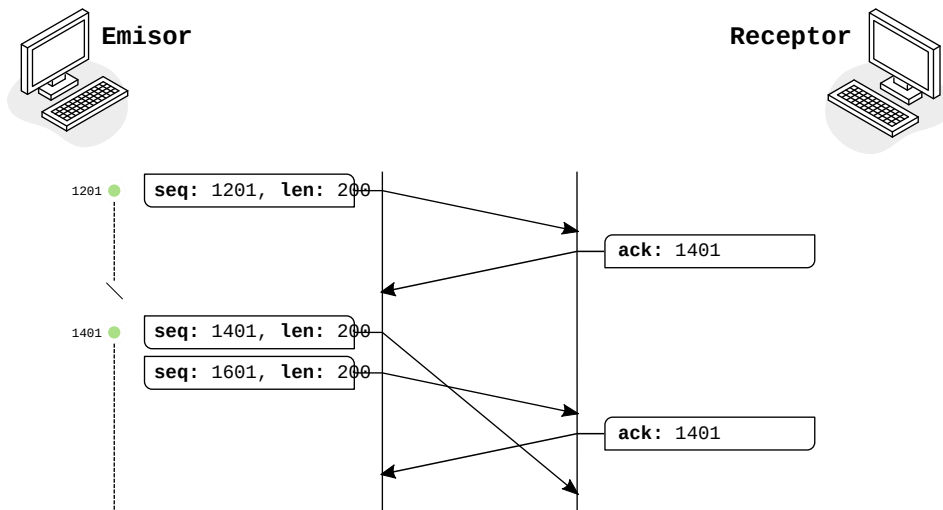


FIGURA 12.13: TCP: Confirmación duplicada

12.12.6. Demasiadas retransmisiones

Si durante el transcurso de una conexión, se suceden las retransmisiones sin lograr confirmar nuevos datos es una señal clara de que algo va realmente mal. Se definen dos umbrales que implican medidas distintas (ver [29]). Si el número de retransmisiones sobrepasa R1, que tiene normalmente un valor de 3, se entiende como un indicio de que podría haber un problema en el rutado de paquetes y se informa al proceso. Si sobrepasa R2, que suele tener un valor de 15, se cierra la conexión. En el caso de Python, eleva una excepción `OSError` con el valor `errno.ETIMEDOUT`.

El Linux puedes consultar estos valores como parámetros del núcleo (§2.10): R1 se corresponde con `tcp_retries1` y R2 con `tcp_retries2`. En el Listado 12.14 muestra cómo hacerlo y puedes ver los valores por defecto.

```
$ sysctl net.ipv4.tcp_retries1
net.ipv4.tcp_retries1 = 3
$ sysctl net.ipv4.tcp_retries2
net.ipv4.tcp_retries2 = 15
```

LISTADO 12.14: TCP: Configuración de los umbrales de retransmisión

Estos valores raramente se modifican. Solo ocurre en situaciones muy concretas, con enlaces de muy alta latencia o mucha inestabilidad, como enlaces vía satélite.

12.13. *Keep alive*

La conexión TCP puede quedar abierta durante mucho tiempo sin que ninguno de los dos extremos envíe segmentos (datos ni confirmaciones). Es estas condiciones se dice que la conexión está «inactiva» (*idle*). Para asegurarse de que el otro extremo sigue ahí, TCP proporciona un mecanismo llamado *keep alive* (mantener vivo) que consiste en enviar un segmento «sonda» para provocar una respuesta (una confirmación) del otro extremo y así verificar que sigue «vivo».

El segmento sonda se envía cuando expira el temporizador *keep-alive* (2 horas). El temporizador se reinicia cada vez que se recibe un segmento. Si el otro extremo no responde a la sonda, el emisor repetirá la operación cada 75 segundos hasta un máximo de 10 intentos. Si a pesar de eso no recibe respuesta, el emisor cierra la conexión. Todos estos valores son configurables por el programador para cada conexión.

- `SO_KEEPAIVE`: Activa o desactiva el mecanismo *keep alive*.
- `TCP_KEEPIIDLE`: Tiempo de inactividad antes de la primera sonda.
- `TCP_KEEPIINTVL`: Intervalo entre sondas.

- `TCP_KEEPCNT`: Número máximo de intentos antes de cerrar la conexión.

El Listado 12.15 muestra cómo fijar estos valores en un programa Python.

```
## Activar keep-alive
sock.setsockopt(socket.SOL_SOCKET, socket.SO_KEEPALIVE, 1)

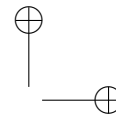
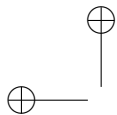
# Configurar el temporizador keep-alive (en segundos)
sock.setsockopt(socket.IPPROTO_TCP, socket.TCP_KEEPIIDLE, 3600)
```

LISTADO 12.15: TCP: Configuración del mecanismo *keep alive*

Y ¿qué más?

Hemos visto el control de flujo que, junto al control de errores, son los mecanismos fundamentales que proporcionan confiabilidad a TCP y por tanto a la mayoría de las comunicaciones en Internet en la actualidad.

Sin embargo, queda al menos otra gran cuestión que abordar: la congestión. Se trata de un problema más complejo que el control de errores, porque no afecta solo a una conexión concreta, sino a una parte o a toda la interred. En el próximo capítulo veremos cómo TCP aporta soluciones para paliar este problema.



Capítulo 13

Control de congestión

Al terminar este capítulo, entenderás:

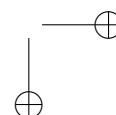
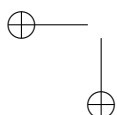
- Qué es la congestión y cuáles son sus causas.
- Por qué la congestión es un problema grave.
- Qué técnicas se pueden aplicar para prevenir o mitigar la congestión.
- Qué técnicas de control de congestión utiliza TCP y cómo funcionan.

Intuitivamente una congestión es una situación en la que la demanda de recursos supera la capacidad del sistema para proporcionarlos. En el caso de las redes de conmutación de paquetes, la congestión se produce cuando la carga, es decir, la cantidad de paquetes que se introduce en la red, supera su capacidad para manejarlos adecuadamente y llevarlos a su destino.

En este capítulo veremos las causas de la congestión, sus consecuencias y las técnicas que se pueden aplicar para prevenirla o bien controlarla una vez que se ha producido.

13.1. Carga, capacidad y congestión

Podemos cuantificar la **carga** de la subred como la cantidad de paquetes que se encuentran en tránsito en un determinado momento. La subred está formada por líneas de comunicaciones y dispositivos de intercambio, de modo que los bytes que forman los paquetes pueden estar físicamente viajando a través de un enlace o bien en las colas de entrada y salida de los routers. Desde este punto de vista, podemos entender la subred como una especie de almacenamiento temporal, una ‘memoria’ que contiene paquetes durante el tiempo que transcurre desde emisor los envía envían hasta que son entregados a su destino. La cantidad máxima de paquetes que cabe en esta ‘memoria’ la podemos entender, por analogía, como la **capacidad** de la subred.



Por simplicidad, hemos hablado de paquetes, pero obviamente esos paquetes son de tamaño muy variado, y lógicamente para una misma cantidad de paquetes, la carga y la capacidad son mayores si hablamos de paquetes de mayor tamaño. Así que, para obtener un cálculo más preciso de carga y capacidad, habría que medirlas en bytes.

Una vez están claros estos conceptos, podemos entender la **congestión** como toda situación en la que la carga supera la capacidad de la subred (ver fórmula 13.1), es decir, se produce una *sobrecarga* de la subred.

$$\text{congestión} = \text{carga} > \text{capacidad} \quad (13.1)$$

Hemos hablado de las colas de entrada y salida de los routers, pero es necesario analizar esta cuestión con detalle. Los paquetes llegan al router a través de las interfaces de red y se almacenan en las colas de entrada, asociadas a cada interfaz. El componente del router que realiza el reenvío (ver § 8.1) va tomando y procesando paquetes desde la cabeza de cada cola. Eso significa que cuantos más paquetes entran al router, mayor es la ocupación de las colas de entrada y más tiempo pasan los paquetes en ellas, lo que aumenta la latencia. Si cuando llega un nuevo paquete, la cola ya está llena, el router lo descarta¹. Puedes ver una representación de esta situación en la Figura 13.1. La cola de entrada de la interfaz superior está llena y no puede aceptar nuevos paquetes.

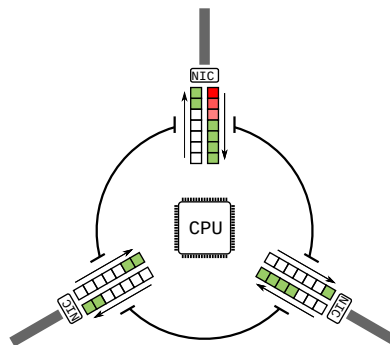


FIGURA 13.1: Esquema de las colas de entrada y salida de un router

Una vez procesado, el paquete se coloca en la cola de salida asociada a la interfaz de red por la que será enviado. La interfaz de salida necesita cierto tiempo para enviar un paquete, por lo que si la velocidad con la que el router reenvía paquetes por esa interfaz (los encola para su envío) es

¹A menos que el router aplique una política de descarte selectivo

mayor que la velocidad con la que se transmiten al medio, la cola de salida también se puede llegar a llenar y también habrá paquetes descartados por ese motivo.

Como puede verse, cuando la carga se acerca o supera la capacidad de la subred, los routers empezarán a descartar paquetes, que es precisamente la consecuencia más grave de la congestión. Pero hay otros efectos negativos, como el incremento de la latencia, que pueden ocurrir mucho antes. La Figura 13.2 muestra cómo la latencia aumenta con la carga y el impacto que tiene sobre el rendimiento al acercarse la carga a la capacidad de la subred. El rendimiento puede medirse como la cantidad de paquetes que la subred es capaz de llevar a su destino por unidad de tiempo.

Quizá pienses que causa es que las colas del router son demasiado pequeñas. Sin embargo, aumentar el tamaño no soluciona el problema. Si la cola es mayor, los paquetes pasarán más tiempo en ella, aumentando la latencia del flujo y provocando la expiración de los temporizadores de retransmisión, cuando se trate de tráfico confiable. Incluso obviando ese efecto indeseado, mayores colas solo retrasarían el momento en que la congestión se hace patente.

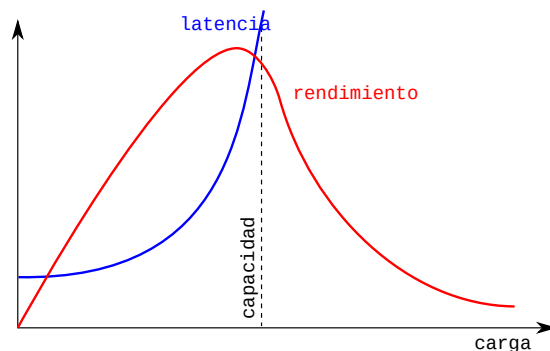


FIGURA 13.2: Carga, capacidad, latencia y rendimiento

En todo caso, la congestión raramente ocurre a la vez en toda la subred. Suele localizarse en algunos routers o enlaces involucrados en determinadas rutas con más uso en ese momento. Mientras tanto quizá en otras partes todo va bien, aunque de no tomarse medidas, la congestión puede propagarse rápidamente.

13.2. Control de congestión

Por supuesto, la congestión es un problema muy grave que puede inutilizar parte de una interred. La solución es lo que llamamos **control de congestión**, que consiste en aplicar técnicas que permitan evitar, o al menos minimizar, los efectos de la congestión.

Veamos una breve descripción de algunas de las técnicas habituales agrupadas en dos categorías: preventivas y reactivas. En ambos casos se basan en la idea de reducir la tasa de emisión de mensajes, o en menor medida, el tamaño de estos. Reduciendo la cantidad de paquetes que se introduce en la red, lógicamente se reduce la carga y con ello la posibilidad de alcanzar la capacidad. Sin embargo, recuerda que el rendimiento óptimo de la subred está cerca de la capacidad, de modo que el objetivo es buscar el equilibrio, llegar al punto de rendimiento máximo sin sobrepasarlo.

13.2.1. Técnicas preventivas

Como su nombre indica, se aplican para evitar que la congestión llegue a producirse. Tanto los emisores como los receptores pueden colaborar. También se llaman *técnicas de bucle abierto* porque no se realizan medidas ni se obtiene información explícita de lo realmente ocurre en la red.

Política de retransmisión

Cuando un paquete se pierde, el emisor puede aumentar la duración del temporizador de retransmisión, para que en una situación de congestión, los paquetes se retransmitan más tarde de lo habitual, reduciendo así la tasa de emisión.

Política de confirmación

Si el receptor estima una situación de congestión puede retrasar el envío de mensajes de confirmación, lo que indirectamente reduce la tasa de emisión. Obviamente este retardo adicional debe estar dentro de los límites que eviten retransmisiones innecesarias, que provocarían un aumento de carga, agravando la congestión. También se puede aprovechar la *confirmación acumulativa* que además reduce el número de mensajes en tránsito.

Política de ventana

Un protocolo de confiabilidad de tipo *repetición selectiva* o *confirmación negativa* evita tráfico innecesario y puede ayudar a evitar la congestión.

Política de descarte

Los routers pueden decidir qué paquetes resulta más conveniente descartar en caso de congestión. Por ejemplo, descartar paquetes que corresponden a protocolos confiables provocará retransmisiones, y eso no ayuda a resolver el problema. Pueden ser técnicas muy efectivas, pero requieren conocimiento detallado de la topología de la red y de los protocolos de aplicación.

Política de admisión

Determina si la red dispone de recursos suficientes para aceptar un nuevo flujo de datos, rechazándolo si no fuera así. Esta técnica es tremendamente efectiva, pero solo es posible en tecnologías de conmutación de circuitos o circuitos virtuales.

13.2.2. Técnicas reactivas

Se aplican una vez que la congestión ha empezado a producirse y algunos de sus síntomas son perceptibles. Se denominan también *técnicas de bucle cerrado* porque se basan en la retroalimentación de información sobre el estado de la red que procede, bien de los receptores, o de otros dispositivos intermedios.

Backpressure

Se trata de una técnica típica de las redes de circuitos virtuales. Cuando un dispositivo de interconexión (conmutador según su terminología²) detecta congestión, puede enviar un mensaje al conmutador inmediatamente anterior para que reduzca su tasa. Esto implicará el descarte de mensajes, de modo que este conmutador enviará a su vez un mensaje de notificación de congestión al conmutador previo y así hasta llegar al dispositivo emisor. Este último reducirá su tasa de emisión y la congestión se irá reduciendo progresivamente.

Paquete de estrangulamiento (*choke*)

Es similar a *backpressure*. Se envía un mensaje específico de notificación de congestión directamente al emisor, mientras que los dispositivos intermedios no son notificados.

Señalización implícita

En este caso el emisor interpreta ciertos indicios en el comportamiento del flujo o conexión como señales de que está ocurriendo un episodio de congestión. Uno de estos indicios podría ser la ausencia o retraso en los mensajes de confirmación.

²No confundir con conmutadores Ethernet

Señalización explícita

Es similar al paquete *choke*, pero no se utiliza un mensaje de notificación de congestión específico, sino que se utilizan campos de los propios mensajes de datos. En algunas tecnologías, como en Frame Relay, esta información podía enviarse tanto al emisor (*backward signaling*) como al receptor (*forward signaling*).

13.3. Control de flujo vs. control de congestión

Ambos, control de flujo y control de congestión se basan en ajustar la tasa a la que nuevos datos entran en la red. Sin embargo, su objetivo es muy diferente. El control de flujo (de receptor) intenta evitar que un emisor específico sature a un receptor específico, es decir, es un problema que atañe únicamente a un único flujo de datos (normalmente una conexión). Por otro lado, el control de congestión busca evitar que una subred o parte de ella colapse por sobrecarga de tráfico, es decir, la congestión afecta a muchos flujos de datos simultáneamente, y para resolverla probablemente múltiples emisores tendrán que colaborar reduciendo sus tasas de emisión.

Como son dos problemas independientes pueden ocurrir de forma independiente. Pueden darse situaciones en las que un emisor tenga que aplicar control de flujo porque está saturando a su receptor, y sin embargo, la red se encuentre en condiciones de baja carga. También puede ocurrir lo contrario: que un emisor tenga que aplicar control de congestión porque la red está saturada, pero su receptor no tenga ningún problema con la tasa de llegada de los datos. Y obviamente también pueden coincidir ambos problemas al mismo tiempo.

13.4. Supresión al origen

El protocolo IPv4 ofrecía un mecanismo para control de congestión muy rudimentario mediante el mensaje ICMP *source quench*, que se podría traducir como «enfriamiento en origen».

Cuando un router IP tiene un alto nivel de ocupación en una cola de entrada (señal inequívoca de congestión) y se ve obligado a descartar un paquete, puede enviar un paquete *source quench* al emisor para que reduzca su tasa de emisión. El mensaje incluye como carga útil los primeros 8 bytes (ver § 8.6) del paquete descartado, de modo que el emisor puede identificar a qué flujo corresponde. Conforme a la clasificación anterior, el uso de este tipo de notificación se puede entender como una técnica reactiva similar al *choke packet*.

Un receptor también puede enviar mensajes *source quench* si no puede manejar la tasa de recepción de datos, de modo que en ese caso funciona como un sistema de control de flujo, no de congestión.

Desde hace años el uso de paquetes *source quench* se considera obsoleto por varias razones [15, 30], y de hecho, no se incluyó ningún mecanismo equivalente en IPv6. Las razones más importantes son las siguientes:

- Su eficacia es limitada si el emisor recibe el mensaje tarde, o simplemente el router no lo envía.
- El mecanismo no incluye una medida de la gravedad de la congestión. El emisor debe decidir cuánto y durante cuánto tiempo reducir la tasa después de recibir las notificaciones.
- Un agente malicioso podría enviar notificaciones falsas para reducir la tasa de una víctima consiguiendo así una denegación de servicio (DoS).
- Los mecanismos de control de congestión de TCP son mucho más eficaces.

13.5. Control de congestión en TCP

TCP cuenta con mecanismos de control de congestión muy sofisticados. Se trata de mecanismos de «caja negra», es decir, los elementos intermedios de la red, como los routers, no participan y no tienen que hacer nada específico para que el control de congestión de TCP funcione correctamente.

El concepto clave es la *ventana de congestión* (*congestion window*), abreviado como *cwnd*. La ventana de congestión limita en todo momento la ventana de envío, es decir, controla la tasa de emisión de cada conexión. Claramente esta es una técnica de «política de ventana» según la clasificación que hemos visto en §13.2. Por supuesto, el valor de la ventana de congestión no es fijo, sino que se recalcula continuamente para adaptarse a las condiciones de la red. Con esto podemos refinar el modo en que se calcula el valor la ventana de envío (ver fórmula 13.2), en contraposición a la versión simplificada que dimos en la fórmula 12.1.

$$swnd \leq \min(rwnd, cwnd) \quad (13.2)$$

Esta fórmula tan sencilla dice algo interesante: ambos mecanismos, control de flujo y control de congestión, son igualmente importantes y tienen la misma capacidad de limitar la tasa de emisión.

La forma en la que se determina el valor de la ventana de congestión y el comportamiento de las retransmisiones depende de 5 algoritmos:

- Arranque Lento (*Slow Start*)
- Evitación de Congestión (*Congestion Avoidance*)
- Decrecimiento multiplicativo (*Multiplicative Decrease*)
- Retransmisión Rápida (*Fast Retransmit*)
- Recuperación Rápida (*Fast Recovery*)

Veamos cada uno de ellos en las siguientes secciones.

13.5.1. Arranque lento (*slow start*)

El *arranque lento* pretende alcanzar una tasa de emisión significativa partiendo de la mínima, pero con un margen de seguridad que evite provocar congestión. Se aplica en el inicio de la conexión o justo después de un RTO. Aunque el arranque lento es solo una de las fases, es habitual encontrar bibliografía que se refiere a todo el control de congestión de TCP como «arranque lento», probablemente porque en las primeras versiones de TCP era de hecho el único algoritmo que había.

El valor de la ventana de congestión se actualiza en base a rondas. Una «ronda» es el tiempo que transcurre desde que se envían los segmentos asociados a una ventana hasta que comienzan a llegar sus respectivos ACKs, lo que normalmente equivale a la medición de un RTT.

Inmediatamente después de establecerse la conexión, se fija el tamaño de la ventana de congestión (*initial window*) en MSS³. Así, el emisor puede empezar enviando un segmento de tamaño MSS. Al llegar su correspondiente confirmación, *cwnd* crece hasta 2 MSS. En la segunda ronda puede enviar 2 segmentos de tamaño MSS. Sus respectivas confirmaciones aumentan *cwnd* hasta 4 MSS, y así sucesivamente. Puedes ver la evolución de *cwnd* en la Figura 13.3.

Este crecimiento exponencial (2^r , siendo r la ronda) continúa hasta que alcanza *ssthresh* (abreviatura de *slow start threshold*) o bien se detecta congestión. Inicialmente *ssthresh* se fija en un valor alto, habitualmente el tamaño máximo de ventana (2^{16}), pero cuando se detecte congestión, se reducirá sensiblemente.

³Esto es una simplificación —de las muchas que aplicaremos en este capítulo— pero sirve para el propósito del libro. Como referencia, las versiones actuales de GNU/Linux suelen fijar un mínimo de *cwnd*=10 MSS.

Es necesario aclarar que el crecimiento tal como se acaba de explicar solo ocurre si $rwnd$ no es un limitante, es decir, siempre que se cumpla $rwnd > cwnd$. En otro caso, la tasa de emisión estará condicionada por $rwnd$ que a su vez, limitará el crecimiento de $cwnd$. Si el emisor envía un segmento de, por ejemplo, $MSS/2$ debido a un valor de $rwnd$ bajo, la confirmación correspondiente aumentará $cwnd$ únicamente en esa cantidad.

En resumen, durante el arranque lento, $cwnd$ aumenta la cantidad de nuevos bytes (N) enviados por cada mensaje de confirmación, siempre que esa cantidad no supere MSS (ver fórmula 13.3). Si efectivamente, los segmentos son de tamaño MSS , $cwnd$ crece a un ritmo MSS por cada confirmación efectiva, o lo que es lo mismo, $cwnd$ se duplica al final de cada ronda.

$$cwnd += \min(N, MSS) \quad (13.3)$$

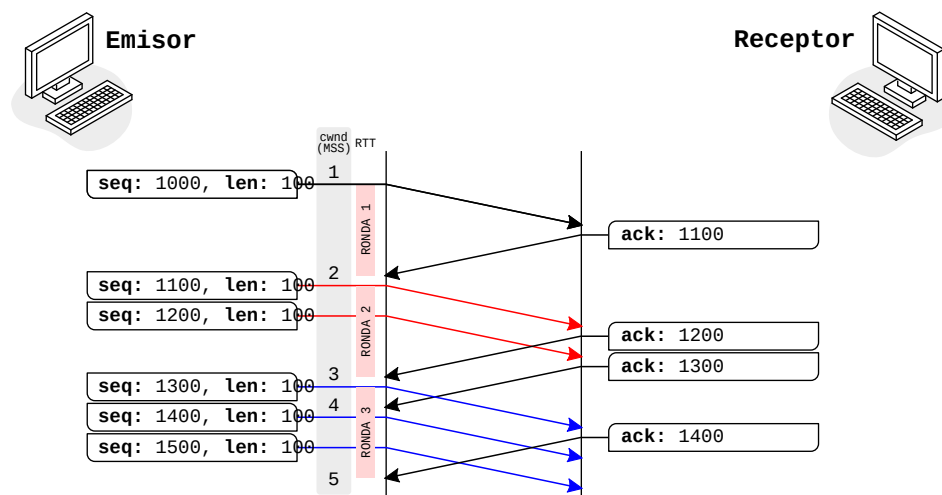


FIGURA 13.3: Evolución de la ventana de congestión durante una fase de arranque lento

Cuando $cwnd > ssthresh$, la fase de arranque lento termina y comienza una fase de *evitación de congestión*.

13.5.2. Evitación de congestión (*congestion avoidance*)

En esta fase $cwnd$ crece de forma lineal según la fórmula 13.4 con cada confirmación neta recibida, que corresponde aproximadamente a una MSS/k siendo k el número de segmentos enviados en esa ronda. Dicho de otro modo, al final de cada ronda, $cwnd$ habrá crecido aproximadamente MSS , siempre

que como se indicó antes, $rwnd$ no sea un limitante. La Figura 13.4 muestra un ejemplo de esta fase.

$$cwnd += MSS^2 / cwnd \quad (13.4)$$

En este caso no se aplica ningún umbral, la ventana continúa creciendo indefinidamente hasta que se detecte algún problema. Sin embargo, el valor de $cwnd$ respecto a $ssthresh$ se puede utilizar para determinar en qué fase se encuentra la conexión:

- Si $cwnd < ssthresh$, la conexión está en una fase de arranque lento.
- Si $cwnd > ssthresh$, la conexión está en una fase de evitación de congestión.
- Si $cwnd = ssthresh$, el emisor puede decidir qué algoritmo es más conveniente en cada caso, es decir, en ocasiones puede continuar con arranque lento, mientras que en otras puede cambiar a evitación de congestión.

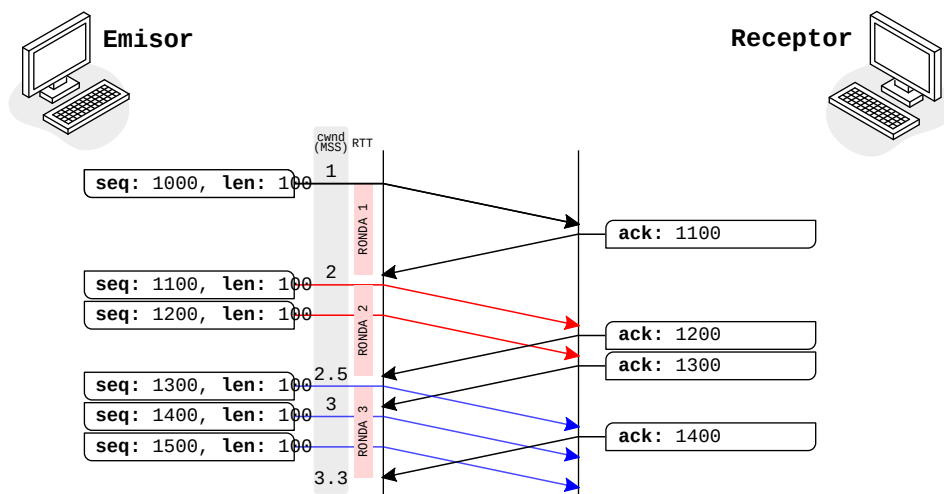


FIGURA 13.4: Evolución de la ventana de congestión durante una fase de evitación de congestión

13.6. Decrecimiento multiplicativo

Esta fase, como indica su nombre, es una reducción drástica de la ventana de congestión que se aplica cuando se detecta congestión. Aunque estaba

diseñada para limitar la tasa durante cierto tiempo, en la actualidad solo se aplica durante una ronda, por lo que en realidad actúa más como transición que como fase estable.

13.6.1. Indicios de congestión

Aunque hemos hablado varias veces de ‘detección de congestión’, en realidad TCP asume que la conexión está sufriendo los efectos de un episodio de congestión debido a dos *indicios* muy concretos:

- Ha expirado el RTO. TCP asume que la explicación más probable de la expiración del RTO y la correspondiente pérdida de un segmento es que un router ha descartado el paquete debido a la alta ocupación de sus colas, señal inequívoca de congestión.
- El emisor ha recibido 3 confirmaciones duplicadas (*3dupack*⁴), es decir, han llegado un total 4 mensajes de confirmación idénticos desde un receptor al que se le están enviando datos nuevos. Las confirmaciones duplicadas se producen porque el receptor está recibiendo segmentos correctos, pero fuera de secuencia (ver § 12.12.5). La Figura 13.5 muestra un ejemplo. Por supuesto, un emisor podría recibir más de 3 en una ronda, dependiendo de cuántos segmentos la formen.

La detección de la congestión en base a estos indicios se puede interpretar como una técnica de ‘señalización implícita’ según la clasificación de § 13.2. Estos indicios tienen consecuencias diferentes. La expiración del RTO es un evento más grave y es más probable que sea síntoma de congestión. La aparición de *3dupack* en cambio indica que los mensajes siguientes, correspondientes a la ronda, siguen llegando al destino. Por tanto, la congestión, de haberla, no parece tan grave.

Cuando ocurre una expiración del RTO, la fase en curso termina inmediatamente. Se inicia entonces una nueva fase de arranque lento, y se reajustan los valores de *cwnd* (fórmula 13.5) y *ssthresh* (fórmula 13.6) del siguiente modo.

$$cwnd = MSS \quad (13.5)$$

$$ssthresh = \max(2MSS, \frac{\text{datos en vuelo}}{2}) \quad (13.6)$$

⁴En muchos documentos técnicos, o en código fuente, es habitual encontrar «dupack» para indicar las 3 confirmaciones duplicadas. Aquí utilizaremos el más explícito «3dupack» para evitar cualquier confusión.

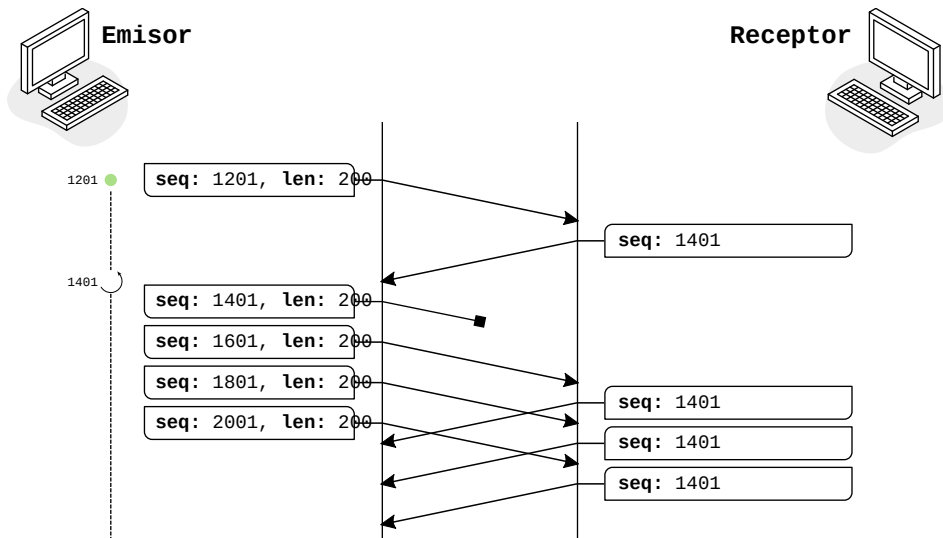


FIGURA 13.5: 3 ACK duplicados, como consecuencia de la pérdida (o retraso) de un segmento

El nuevo valor de *ssthresh* provoca que, en la siguiente fase de arranque lento, el crecimiento de *cwnd* sea más conservador que en el inicio de la conexión. Por eso, la tasa de emisión se adapta con más suavidad a las condiciones de la red. El parámetro *datos en vuelo* (*FlightSize*) se refiere a la cantidad de datos que el emisor ha enviado y que todavía no han sido confirmados.

Cuando se produce *3dupack*, la fase en curso termina. En ese momento se aplica el algoritmo de ‘retransmisión rápida’ (*fast retransmit*) que inmediatamente realiza una retransmisión del segmento al que corresponde la confirmación duplicada. Se llama así porque no espera la expiración del RTO.

Fíjate que la aparición de los ACK duplicados no implica necesariamente que el mensaje se haya perdido. Podría acabar llegando y el receptor enviaría una confirmación que, de llegar a tiempo, evitaría la retransmisión.

Después de la retransmisión rápida se aplica el algoritmo de ‘recuperación rápida’ (*fast recovery*). Primero fija *cwnd* y *ssthresh* según la fórmula 13.6. Después lleva a cabo un período de «inflación» que incrementa *cwnd* en 3 MSS para compensar las confirmaciones duplicadas. A partir de ahí, incrementa *cwnd* en MSS por cada confirmación duplicada adicional. Durante este período, el emisor puede enviar segmentos con datos nuevos. Cuando por fin se recibe la confirmación para el segmento retransmitido, se produce la «deflación», es decir, *cwnd* vuelve al valor *ssthresh* calculado previamente. La conexión continua en una fase de evitación de congestión.

Los mecanismos de retransmisión y recuperación rápida fueron optimizaciones introducidas en las versiones Tahoe y Reno, respectivamente. Antes de eso (versión Vegas), al recibir *3dupack*, el emisor cambiaba directamente a una fase de evitación de congestión fijando *cwnd* a *swnd*/2, algo que más tarde se consideró una caída demasiado brusca de la tasa de envío.

La Figura 13.6 muestra el diagrama de transición entre fases según lo explicado.

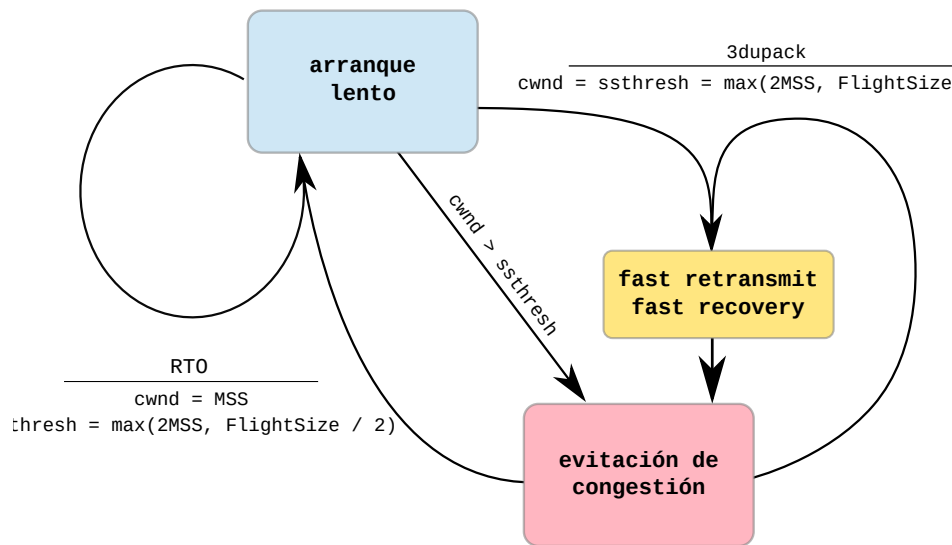


FIGURA 13.6: Transición entre los algoritmos de control de congestión de TCP

13.6.2. Modelo simplificado

Por facilitar el estudio y comprensión de estos mecanismos podemos aplicar una serie de simplificaciones que no afectan significativamente a su finalidad, pero que nos permiten realizar cálculos más sencillos.

En la fase de evitación de congestión, en lugar de aplicar la fórmula 13.4 para determinar el crecimiento de *cwnd*, utilizaremos la fórmula 13.7, que divide el crecimiento de *MSS* de la ronda en los *k* segmentos que se envían en ella, con lo que al final de la ronda *cwnd* también habrá crecido *MSS*.

$$cwnd+ = MSS/k \quad (13.7)$$

Cuando expira el *RTO*, calcularemos el nuevo valor de *ssthresh* según la fórmula 13.6 que utiliza *swnd* como aproximación a *datos en vuelo*. También

obviamos el mínimo de 2 MSS. El cálculo simplificado es el de la fórmula 13.8.

$$ssthresh = swnd/2 \quad (13.8)$$

Como parte de la simplificación, aplicaremos el enfoque de TCP Vegas, es decir, después de *3dupack* se aplica la fórmula 13.9 y se pasa a una fase de evitación de congestión, obviando la retransmisión rápida y la recuperación rápida.

$$cwnd = swnd/2 \quad (13.9)$$

La Figura 13.7 muestra el diagrama de estados entre las fases arranque lento y evitación de congestión, aplicando la simplificaciones indicadas.

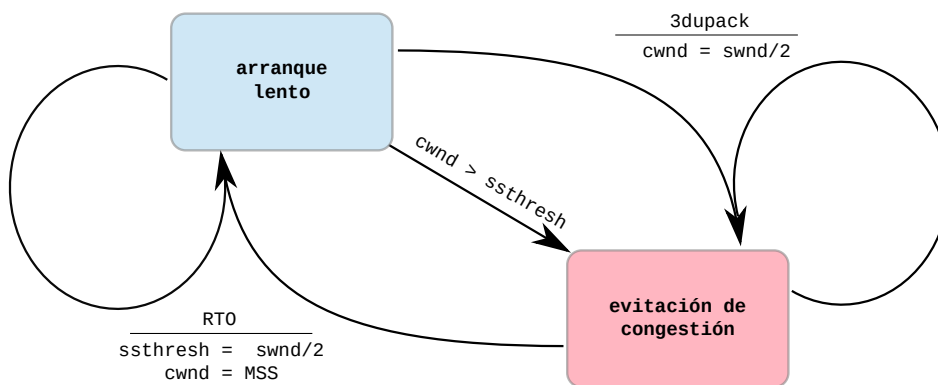


FIGURA 13.7: Transición entre los algoritmos de control de congestión de TCP

En la Figura 13.8 aparece un ejemplo más completo para entender cómo evoluciona la ventana de congestión conforme ocurren diferentes eventos a lo largo de la conexión. El valor de $cwnd$ aparece en el eje vertical expresado en MSS mientras que el eje horizontal muestra las rondas, o períodos RTT equivalente. Por simplicidad este ejemplo supone que $rwnd > cwnd$ durante toda la conexión, de forma que $swnd = cwnd$.

Veamos cómo progresa la ventana de congestión respondiendo a los indicios que ocurren durante la conexión:

Ronda 0 – Al inicio de la conexión hay siempre una fase de arranque lento (AL) que fija $cwnd = MSS$ y lo duplica en cada ronda.

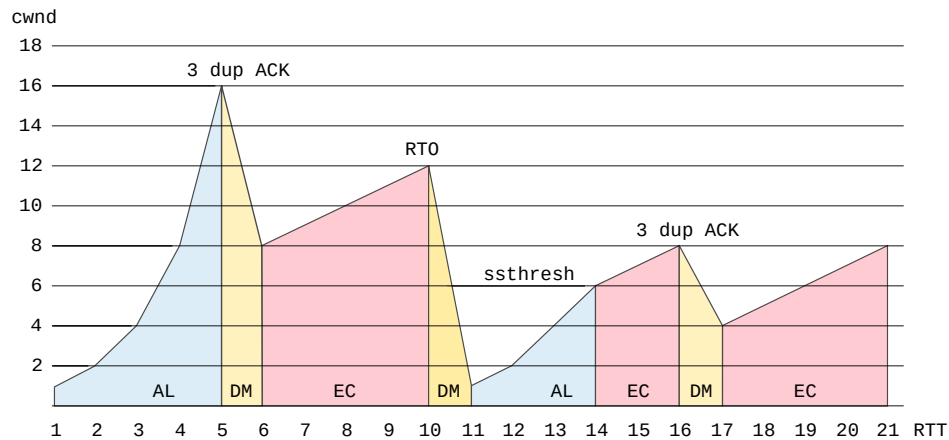


FIGURA 13.8: Ejemplo de evolución de la ventana de congestión en TCP

Ronda 4 – Se detecta una situación 3dupack. TCP responde según el diagrama de la Figura13.7 fijando $cwnd = swnd/2 = 16/2 = 8MSS$ con una transición de disminución multiplicativa (DM) y cambia a evitación de congestión (EC). La fase EC incrementa el valor de cwnd en MSS en cada ronda.

Ronda 9 – Aparece otro problema: la expiración del RTO. Esta vez la respuesta es más agresiva, cwnd se reduce a MSS y comienza una nueva fase de arranque lento, pero también se ajusta $ssthresh = swnd/2 = 12/2 = 6MSS$.

Ronda 13 – El valor de cwnd alcanza $ssthresh$, por lo que entra en evitación de congestión.

Ronda 15 – De nuevo otra situación 3dupack, y de nuevo se aplica $cwnd = swnd/2 = 8/2 = 4MSS$. A partir de aquí cwnd sigue creciendo de forma lineal hasta la ronda 20.

13.7. Gestión activa de colas

Que los routers descarten paquetes cuando sus colas de entrada desbordan tiene algunas consecuencias negativas adicionales a la propia pérdida de los paquetes. Es muy probable que paquetes que corresponden a conexiones diferentes lleguen al router mientras la cola está llena, y todos ellos sufrirán la misma suerte. Esto puede provocar que esas conexiones *sincronicen* sus retransmisiones, un efecto que no es deseable en absoluto.

La Gestión Activa de Colas (en inglés AQM) es un conjunto de técnicas para detectar la congestión antes de que las colas de entrada del router se llenen por completo y, en ese caso, informar de la situación a los nodos finales.

En Internet, este tipo de notificaciones se realizan mediante Notificación Explícita de Congestión (en inglés ECN) que, como su nombre indica, es una técnica de bucle cerrado. Los routers AQM no crean mensajes específicos sino que etiquetan con la notificación Congestion Experienced (CE) la cabecera de los paquetes IP en tránsito. Estos paquetes «marcados» siguen su camino del modo habitual hacia su destino.

Con AQM, los routers tienen dos opciones al detectar una congestión inminente. Pueden *notificar* la congestión de forma implícita: descartando un paquete aunque no haya desbordamiento de la cola de entrada, o de forma explícita: haciendo llegar información específica sobre la congestión al emisor responsable de ese tráfico, es decir, aplicando el marcado ECN.

Con AQM la congestión no se notifica cuando la cola desborda, sino que se utiliza un algoritmo como RED. Este algoritmo calcula la ocupación media de la cola en cierto período de tiempo. Si ese valor sobrepasa determinado umbral, considera que existe un riesgo significativo de congestión y optará por descartar un paquete de la cola o bien aplicar ECN. Es interesante destacar que incluso en el caso de descarte de paquetes, los efectos negativos no son tan graves como cuando ocurre porque la cola desborda, ya que se reduce mucho la probabilidad no deseada de retransmisiones sincronizadas entre varias conexiones.

AQM utiliza ECN únicamente si la carga útil del paquete es un ECT (ECN Capable Transport), es decir, un protocolo de transporte que soporta ECN. En caso contrario, descarta el paquete. ECN utiliza un campo del mismo nombre que hay en la cabecera del paquete IP⁵. Este campo de 2 bits sirve para que el router notifique CE, pero también para indicar a los routers que el paquete encapsula un ECT. Por concretar el significado de estos bits:

- 00:** El protocolo de transporte no soporta ECN.
- 01:** Es un ECT de tipo 1 (ECT 1), el modelo avanzado.
- 10:** Es un ECT de tipo 0 (ECT 0), el modelo estándar, que usa TCP.
- 11:** Un router ha detectado congestión (CE).

13.7.1. ECN en TCP

Un emisor TCP con soporte ECN reacciona a una notificación CE reduciendo su ventana de congestión a la mitad ($cwnd = cwnd/2$) [31], de forma similar a lo que ocurre con la detección de *3dupack*.

⁵Tanto en IPv4 como en la de IPv6

Sin embargo, para que el emisor TCP sea informado se necesita la colaboración del receptor. Para ello se utilizan 2 flags en la cabecera TCP:

ECE (ECN Echo) Lo envía el receptor (probablemente en un segmento de confirmación) para indicar al emisor la llegada de un paquete marcado con CE.

CWR (Congestion Window Reduced) Lo envía el emisor (probablemente en el siguiente segmento de datos) para indicar al receptor que ha recibido la notificación de congestión y ya está reduciendo su tasa de emisión. El receptor puede entonces dejar de marcar sus ACK con el bit ECE.

Para contar con la colaboración del receptor, es necesario que ambos extremos verifiquen que el otro también soporta ECN. Para ello, un cliente que soporta ECN debería activar los flags CWR y ECE en el segmento inicial junto al flag SYN. Si el servidor también soporta ECN, debe responder activando el flag ECE junto con los flags SYN y ACK, propios del proceso de conexión habitual. El uso de estos flags durante el establecimiento de conexión se llama «negociación ECN» y durante su ejecución no tienen el significado de notificación previamente descrito. Solo en el caso en que se haya dado esta negociación ECN con éxito, los participantes marcarán los paquetes IP con ECN, y reconocerán y activarán los flags ECE y CWR durante la conexión.

Veamos una secuencia de mensajes que aparecerían en una conexión que utiliza ECN, que está también representada en la Figura 13.9.

1. El cliente realiza la conexión enviando un segmento marcando los flags SYN, ECE y CWR.
2. El servidor responde con un segmento marcando los flags SYN, ACK y ECE. A partir de este momento, cliente y servidor marcarán los paquetes IP que envíen con el valor ECT(0)⁶.
3. En algún momento, un router detecta congestión y elige un paquete que corresponde a esta conexión. Cambia el valor de su campo ECN a 11, es decir, lo marca como CE.
4. El paquete marcado con CE llega al receptor TCP.
5. En las siguientes confirmaciones, el receptor activa el flag ECE en la cabecera TCP para notificar al emisor.

⁶En realidad, solo marcan los paquetes IP que contengan segmentos TCP con datos.

6. El emisor TCP recibe un segmento con ECE, reduce su tasa de emisión y, en el siguiente segmento de datos, activa el flag CWR.
7. El receptor recibe el segmento marcado con CWR y deja de enviar segmentos marcados con ECE para que el emisor deje de aplicar la reducción de tasa.

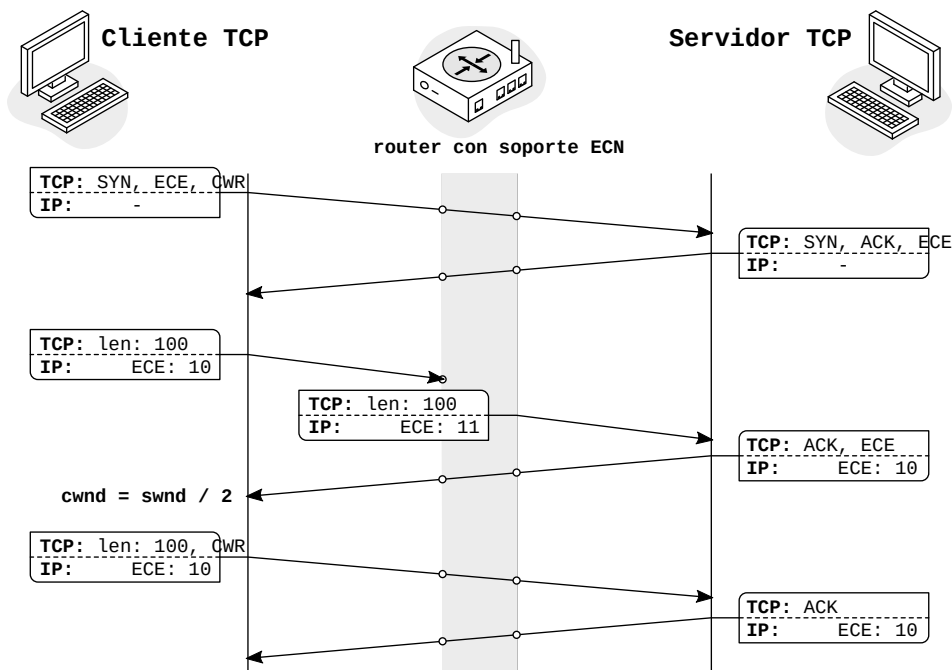


FIGURA 13.9: Conexión TCP con soporte ECN bajo un episodio de congestión

13.7.2. ECN con otros protocolos

ECN no es para uso específico de TCP, si bien requiere que los protocolos de transporte proporcionen la funcionalidad necesaria para realizar la necesaria gestión de tasa de emisión por sus propios medios. Los protocolos de aplicación también pueden beneficiarse de ECN si están diseñados para ello. Por ejemplo, protocolos modernos como QUIC, que de hecho se encapsula en UDP, lo incorporan [32].

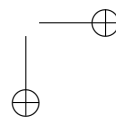
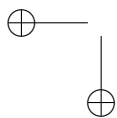
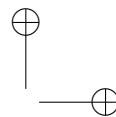
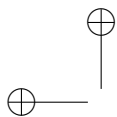
Y ¿qué más?

TCP es un protocolo complejo, lleno de detalles y algoritmos que han sido refinados y optimizados a lo largo de sus muchos años de vida. La expe-

riencia empírica ha permitido a varias generaciones de ingenieros ir adaptando su funcionamiento para trabajar con aplicaciones y servicios que ni se concebían cuando se diseñó a finales de los 70. A pesar de ello, su funcionamiento básico sigue siendo esencialmente el mismo, y su desempeño es francamente robusto para la mayoría de los casos, algo que resulta especialmente meritorio si se piensa por un momento la tremenda evolución que ha experimentado hardware y software desde entonces.

Eso no significa que sea perfecto, ni mucho menos. Algunos de sus problemas más graves son: ausencia total de mecanismos de seguridad, bajo desempeño en comunicaciones móviles, rendimiento pobre en enlaces de alta latencia, el problema del «bloqueo de línea de cabecera» (*head-of-line blocking*), etc.

Estos problemas y sus soluciones son temas avanzados que superan el alcance actual de este libro, pero pueden dar al lector una idea de cómo el estudio y desarrollo de nuevos protocolos de transporte sigue siendo un tema de investigación abierto aún hoy día.



Capítulo 14

Cliente-Servidor

Al terminar este capítulo, entenderás:

- Qué características tiene el modelo cliente-servidor.
- Cómo se crean servidores concurrentes multiproceso o multihilo
- Qué es el *preforking* y el *pool* de procesos e hilos.
- Qué consecuencias tienen las operaciones bloqueantes.
- Cómo utilizar los mecanismos de programación asíncrona para conseguir concurrencia eficiente.

En el capítulo 6 ya vimos las ideas básicas de modelo cliente-servidor como forma de acercarnos a los sockets. Ahora profundizaremos en este modelo y veremos sus posibilidades y limitaciones. Así pues, vamos a continuar en el punto donde lo dejamos en ese capítulo.

El modelo (o paradigma) cliente-servidor (Figura 14.1) es, sin ninguna duda, el enfoque más simple y habitual para crear aplicaciones distribuidas. La mayoría de los **protocolos de aplicación** clásicos de TCP/IP: HTTP, DNS, IMAP, SMTP, etc. están basados en este modelo, y a día de hoy, también la mayoría de aplicaciones que se desarrollan lo siguen usando; tal es el caso de la omnipresente arquitectura REST.

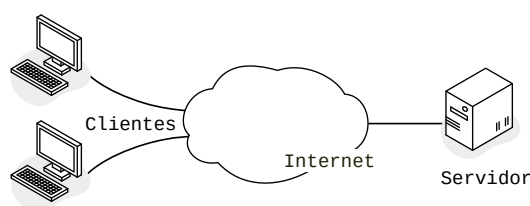


FIGURA 14.1: Modelo Cliente-Servidor

Una *aplicación distribuida* es una que está compuesta por varios componentes que se ejecutan en distintos nodos y colaboran entre sí mediante el

intercambio de mensajes para realizar alguna tarea. En esa definición cabe la gran mayoría de aplicaciones que utilicen la red para comunicarse, que a su vez son la mayoría de las que usamos hoy en día. Sí, casi todo el software que usamos a diario son aplicaciones distribuidas.

En este modelo cliente-servidor aparecen exactamente esos dos **roles** bien diferenciados que le dan el nombre:

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

Que sean ‘roles’ es importante porque un mismo programa puede tomar uno, otro o ambos en distintos momentos o circunstancias, es decir, no definen tipos de aplicaciones. Es cierto que coloquialmente hablamos de ‘servidores’ y ‘clientes’ refiriéndonos a programas concretos: *p. ej.* el servidor web y el cliente de correo. Pero eso se debe simplemente a que uno de los roles destaca muy claramente sobre el otro, y ciertamente también es habitual encontrar programas que solo actúan como cliente.

También es común hablar de nodos como ‘clientes’ y, sobre todo, como ‘servidores’ para denotar que su actividad principal es la de alojar ese tipo de programas. Aunque la arquitectura hardware de un nodo de cómputo puede ser exactamente la misma independientemente de si ejecuta programas servidores o clientes, es fácil identificar algunas características diferenciadoras: el que llamamos ‘servidor’ suele disponer de una mejor conexión a la red, mayor ancho de banda, está montado en un *rack*, tiene fuente de alimentación redundante y no dispone de pantalla, teclado, ratón u otros periféricos habituales, ya que ninguna persona lo va a utilizar como PC de escritorio. Es hardware diseñado para funcionar ininterrumpidamente porque los programas servidores suelen estar pensados para ejecutarse de forma continua e indefinida.

Sin embargo, en un entorno de desarrollo las aplicaciones servidoras se ejecutan en los PC de los desarrolladores o bien en máquinas virtuales o contenedores. Incluso en un entorno doméstico es relativamente común que los usuarios ejecuten en sus máquinas servidores de diversa índole, aunque por supuesto es más habitual que ejecuten clientes.

El modelo cliente-servidor implica un patrón de comunicación característico conocido como **petición-respuesta**. En éste, el cliente realiza una

petición —que suele incluir una operación y un identificador de recurso— y el servidor devuelve una respuesta con un resultado o un valor de retorno, 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 ejemplo, el protocolo HTTP encaja bastante bien en esta idea. Las operaciones son `GET`, `POST`, `PUT` y `DELETE`, etc., y los recursos se identifican con una URL. Por supuesto, existe una gran cantidad de protocolos, pero la mayoría siguen esta pauta. Los recursos que ofrece un servidor pueden ir desde una impresora hasta elementos de menor granularidad, como los registros de una base de datos o los archivos de un directorio. Sin embargo, en este capítulo no vamos a hablar de protocolos de aplicación.

En su lugar, vamos a hablar de productividad y rendimiento. En la mayor parte de las situaciones, el modelo cliente-servidor implica un servidor atendiendo a múltiples clientes simultáneamente. Para lograr eso con una forma razonable, se requiere algún mecanismo de ejecución concurrente, ya sean procesos, hilos, o E/S asíncrona. Todo eso es lo que vamos a tratar en este capítulo.

14.1. Upper

Para concretar las técnicas que vamos a estudiar, utilizaremos un servicio muy básico: un conversor a mayúsculas, es decir, el servidor devuelve una versión en mayúsculas de la cadena de texto que el cliente le envía. Es equivalente al típico servicio *echo*, pero en lugar de simplemente devolver la misma cadena que se le envía, ese pequeño cambio de pasar a mayúsculas que introduce *upper* demuestra que el servidor está haciendo trabajo, por simple que sea. El funcionamiento del cliente es trivial: obtiene cadenas desde la consola, las envía al servidor, recibe las respuestas y las imprime en pantalla.

Para explorar las distintas opciones vamos a empezar por TCP, ya que al estar orientado a conexión, da más juego a la hora de evaluar su rendimiento. Después veremos qué puede hacer UDP y qué diferencias prácticas existen.

14.2. Servidor TCP

En esta primera versión (Listado 14.1) tenemos un bucle al final del listado que acepta conexiones y las delega a una función `handle()` que se encarga de la comunicación con el cliente hasta su desconexión. Este esquema se

llama patrón *acceptor* y es el mismo que vimos en el Listado 6.6 y que se repite en todos los ejemplos del capítulo.

```
import sys
import time
import socket

def upper(msg):
    time.sleep(1) # simulates a complex job
    return msg.upper()

def handle(sock, client):
    print(f"Client connected: {client}")
    while 1:
        data = sock.recv(32)
        if not data:
            break

        sock.sendall(upper(data))

    sock.close()
    print(f"Client disconnected: {client}")

def main(port):
    sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    sock.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
    sock.bind(('', port))
    sock.listen(5)

    while 1:
        conn, client = sock.accept()
        handle(conn, client)

if len(sys.argv) != 2:
    print("Usage: {0} <port>".format(sys.argv[0]))
    sys.exit(1)

try:
    main(int(sys.argv[1]))
except KeyboardInterrupt:
    print("shut down")
```

LISTADO 14.1: Servidor TCP

 /upper/tcp_server.py

En un servidor monoproceso y monohilo (como este) existen dos puntos de bloqueo: esperar un nuevo cliente —en `accept()`—, o esperar un nuevo mensaje del cliente conectado —en `recv()`. y el proceso solo puede estar en uno de ellos. Por supuesto, eso también implica que solo puede haber un cliente conectado; hasta que no acabe el bucle de la función `handle()`, no podrá ejecutar la siguiente iteración del bucle de la función `main()`.

Se le llama servidor *iterativo*, porque *itera* atendiendo cliente tras cliente, propiciando una *serie* de conexiones. Dicho de otro modo: un cliente no

será atendido hasta que el servidor termine con el anterior. Aquí ‘iterativo’ es por definición lo contrario a ‘concurrente’.

Un detalle relevante que hay que comentar es la forma en la que se hace la conversión a mayúsculas mediante la función `upper()`. Aparte de llamar al método `bytes.upper()`, que es la que realmente hace el trabajo, invoca `time.sleep(1)` para provocar una pausa artificial de 1 segundo. El motivo es simular que pasar a mayúsculas es una tarea compleja, para así poder apreciar más fácilmente el efecto de las técnicas de concurrencia que veremos después y la mejora que suponen.

Otro detalle que te puede llamar la atención es la llamada a `setsockopt()`. Como su nombre indica sirve para modificar el valor de una *opción* del socket. Estas opciones sirven para modificar ciertos comportamientos del socket. En concreto esta (`SO_REUSEADDR`) permite arrancar un servidor en un puerto que había sido utilizado recientemente. Esto es solo una conveniencia para los ejemplos y no es recomendable en producción. Veremos este asunto con detalle más adelante en § 12.3.1.

14.3. Cliente TCP

El cliente, después de conectar con el servidor, ejecuta un bucle que lee una línea de texto desde consola —llamada `sys.stdin.readline()`—, la envía al servidor y espera la respuesta.

El tratamiento de la E/S parcial en este caso es simple. Los mensajes que se mueven entre cliente y servidor no necesitan límites definidos, no importa si se trocean o dónde, al final todo el texto llegará al servidor y toda la secuencia de respuestas llegará de vuelta al cliente. En todo caso, para que el funcionamiento resulte más intuitivo en cualquier circunstancia, el cliente se asegura de que recibe una respuesta del mismo tamaño que el último mensaje enviado; en el bucle `while len(msg) < sent` al final de la función `main()`.

```
import sys
from socket import socket

def main(host, port):
    with socket() as sock:
        sock.connect((host, port))

        while 1:
            data = sys.stdin.readline().strip().encode()
            if not data:
                break

            sock.sendall(data)
```

```
msg = bytes()
while len(msg) < len(data):
    msg += sock.recv(32)

print("Reply is '{0}'".format(msg.decode()))

if len(sys.argv) != 3:
    print("Usage: {0} <host> <port>".format(sys.argv[0]))
    sys.exit(1)

try:
    main(sys.argv[1], int(sys.argv[2]))
except KeyboardInterrupt:
    print("shut down")
```

LISTADO 14.2: Cliente TCP

 /upper/tcp_client.py

Para probar el programa simplemente ejecuta servidor y cliente en consolas distintas tal como aparece a continuación. Para terminar la conexión simplemente cierra la entrada estándar del cliente, lo que puedes conseguir pulsando **Ctrl+D**.

Cliente

```
$ ./tcp_client.py
hola
Reply is 'HOLA'
```

Servidor

```
$ ./tcp_server.py 2000
Client connected: ('127.0.0.1', 43172)
Client disconnected: ('127.0.0.1', 43172)
```

FIGURA 14.2: Servidor y cliente upper TCP

Un pequeño detalle al que debes prestar atención es la codificación de los mensajes. La función `readline()` devuelve una cadena de caracteres (`str`) y del mismo modo la función `print()` acepta una cadena. Sin embargo, los métodos `send()` y `recv()` del socket solo aceptan y devuelven secuencias de tipo `bytes`. Por eso es necesario convertir entre estos tipos con los métodos `encode()` y `decode()` respectivamente. Curiosamente el servidor no ha tenido que hacer estas conversiones porque también existe un método `bytes.upper()`, que es el que está usando la función `upper()` del Listado 14.1. Veremos más sobre esta cuestión en el capítulo 17.

Mientras tienes un cliente conectado, puedes ejecutar un nuevo cliente en otra consola, para así comprobar que efectivamente el servidor no acepta esta segunda conexión hasta que la primera termine.

Por cierto, este cliente es tan simple que de hecho lo puedes sustituir directamente con `ncat`, y el funcionamiento es idéntico.

14.4. Servidor TCP multihilo

Un enfoque muy habitual para permitir que el servidor atienda múltiples clientes es crear un hilo (una instancia de la clase `threading.Thread`) en el que ejecutar la función `handle()` para cada nueva conexión. El código completo de este servidor multihilo (*threading server*) es muy similar al servidor iterativo. Lo puedes ver en el Listado 14.3.

```
import sys
import time
import socket
from threading import Thread

def upper(msg):
    time.sleep(1)
    return msg.upper()

def handle(sock, client, n):
    print(f"Client {n:>3} connected: {client}")
    while True:
        data = sock.recv(32)
        if not data:
            break
        sock.sendall(upper(data))

    sock.close()
    print(f"Client {n:>3} disconnected: {client}")

def main(port):
    sock = socket.socket()
    sock.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
    sock.bind(('', port))
    sock.listen(30)

    n = 0
    while True:
        conn, client = sock.accept()
        t = Thread(
            target=handle, args=(conn, client, n := n+1), daemon=True)
        t.start()

    if len(sys.argv) != 2:
        print("Usage: {0} <port>".format(sys.argv[0]))
        sys.exit(1)

    try:
        main(int(sys.argv[1]))
    except KeyboardInterrupt:
        print("shut down")
```

LISTADO 14.3: Servidor TCP multihilo

 `upper/tcp_thread.py`

El programa no limita el número de hilos, aunque por supuesto el SO sí que lo hará. Si quieres detectar un exceso de hilos debes capturar la excepción

`RuntimeError`. Este límite tiene que ver principalmente con la cantidad de memoria que el SO asigna a cada hilo. Los hilos son muy ligeros porque todos ellos comparten el mismo espacio de direcciones (la misma memoria) que el proceso original y solo necesitan memoria adicional para la ‘traza de pila’ (*call stack*). La traza de pila almacena las llamadas a las funciones con sus argumentos, es decir, supone poca memoria, con lo que es posible crear varios miles de hilos sin problema.

Crear y destruir hilos es rápido, de modo que si la tarea que realiza el servidor es sencilla y corta, el servidor multihilo es bastante eficiente.



Python tiene una limitación técnica importante debida al GIL, que impide que varios hilos puedan ejecutar código Python al mismo tiempo, es decir, impide el paralelismo. Esto es un grave problema para programas intensivos en CPU, pero no lo es en absoluto para programas intensivos en E/S, que es precisamente lo que son los servidores.

14.5. Forzando los límites

Para dejar patente la mejora de rendimiento que supone el servidor multihilo, se proporciona un programa llamado `upper/tcp_stress_client.py` que crea la cantidad de clientes que se le solicita y trata de conectarlos al servidor.

En paralelo, cada uno de esos clientes envía 8 palabras. En concreto cada uno envía *twenty tiny tigers take two taxis to town* en 8 mensajes distintos y al terminar, desconecta. Con el siguiente comando puedes probar el cliente de *stress* con el servidor iterativo. Verás que aparecen las 8 respuestas a los mensajes del ‘cliente 0’ (están identificados con un índice entre corchetes) antes de empezar a ver los mensajes del ‘cliente 1’.

Cliente

```
$ ./tcp_stress_client.py 127.0.0.1 2000 10
- [ 0] Reply: TWENTY
- [ 0] Reply: TINY
...
- [ 0] Reply: TO
- [ 0] Reply: TOWN
- [ 1] Reply: TWENTY
- [ 1] Reply: TINY
- [ 1] Reply: TIGERS
...
```

Servidor

```
$ ./tcp_server.py 2000
Client connected: ('127.0.0.1', 47900)
Client disconnected: ('127.0.0.1', 47900)
Client connected: ('127.0.0.1', 42700)
Client disconnected: ('127.0.0.1', 42700)
Client connected: ('127.0.0.1', 42714)
Client disconnected: ('127.0.0.1', 42714)
Client connected: ('127.0.0.1', 42724)
Client disconnected: ('127.0.0.1', 42724)
Client connected: ('127.0.0.1', 42738)
```

FIGURA 14.3: Servidor y cliente upper TCP

Para que los clientes no esperen indefinidamente, el programa fija un *timeout* para la conexión, que de expirar provoca que el cliente desista. Lo verás si lo ejecutas con demasiados clientes.

Si los 10 clientes consiguen conectarse, la ejecución completa necesitará unos 80 segundos (10 clientes $\times$ 8 mensajes $\times$ 1 segundo por mensaje).

Para ver la diferencia con el nuevo servidor, que es la intención, ejecuta ahora el mismo comando con el servidor multihilo de esta sección. En este caso la ejecución se completará en algo más de 8 segundos. Mientras el servidor sea capaz de crear tantos hilos como clientes se conecten a la vez, el tiempo total será aproximadamente esos mismos 8 segundos.

Te recomendamos que pruebes con distintas cantidades de clientes y observes el comportamiento del servidor. Llévalo al límite, prueba con cantidades de clientes cada vez mayores y trata de entender lo que ocurre en cada caso. Y por supuesto, no olvides probarlo también con los servidores que veremos en las siguientes secciones.

14.6. Servidor TCP multiproceso

El servidor multiproceso (*forking server*) es la alternativa mas frecuente al multihilo. El código es más complejo por la necesaria gestión de procesos. El programa en sí también es más costoso que el multihilo. Los procesos son mucho más pesados que los hilos. Implican una copia completa de la memoria del proceso padre. Por otro lado, los procesos son independientes unos de otros, de modo que los errores de uno no afectan a los demás y no hay ningún GIL que afecte a las tareas intensivas en CPU.

Respecto al código, la diferencia clave es que la función `handle()` se ejecuta ahora dentro de un nuevo proceso hijo. Aparte de eso, las funciones `main()`, `handle()` y `upper()` son prácticamente las mismas.

```
import sys
import os
import time
import socket

class ProcessThrottler(object):
    def __init__(self, max_procs=40):
        self.max_procs = max_procs
        self.procs = []

    def collect_children(self):
        # from socketserver module
        while self.procs:
            opts = os.WNOHANG if len(self.procs) < self.max_procs else 0
            pid, status = os.waitpid(0, opts)
```

```

        if not pid:
            break

        self.procs.remove(pid)

    def start_new_process(self, func, args):
        self.collect_children()
        pid = os.fork()
        if pid:
            self.procs.append(pid)
        else:
            func(*args)
            sys.exit()

    def upper(msg):
        time.sleep(1) # simulates a complex job
        return msg.upper()

    def handle(sock, client, n):
        print(f"Client {n:>3} connected: {client}")
        while 1:
            data = sock.recv(32)
            if not data:
                break
            sock.sendall(upper(data))

        sock.close()
        print(f"Client {n:>3} disconnected: {client}")

    def main(port):
        sock = socket.socket()
        sock.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
        sock.bind('', port)
        sock.listen(5)

        throttler = ProcessThrottler()
        n = 0

        while 1:
            conn, client = sock.accept()
            throttler.start_new_process(
                handle, (conn, client, n := n+1))

    if len(sys.argv) != 2:
        print("Usage: {0} <port>".format(sys.argv[0]))
        sys.exit(1)

    try:
        main(int(sys.argv[1]))
    except KeyboardInterrupt:
        print("shut down")

```

LISTADO 14.4: Servidor TCP multiproceso

📄/upper/tcp_fork.py

La creación de los procesos recae exclusivamente en la clase `ProcessThrottler`. La clase guarda una lista de los PID de los procesos que va crean-

do (procs). El método `start_new_process()` es el que ejecuta la llamada `os.fork()` e invoca la función indicada como argumento. El método `collect_children()`¹ se encarga de limitar la cantidad de procesos que se crearán, invocando `os.waitpid()` de forma bloqueante si se alcanza el máximo, evitando así que se cree un nuevo proceso hasta que no haya terminado uno de los existentes. Ese máximo se indica en el constructor con el parámetro `max_procs` (por defecto: 40).

Python dispone de algunas abstracciones de más alto nivel para la creación y gestión de procesos. Por ejemplo, la clase `multiprocessing.Process` ofrece un API similar a la de `threading.Thread` para manejar un único proceso, mientras que `multiprocessing.Pool` y `ProcessPoolExecutor` del módulo `concurrent.futures` permiten crear un *poll*² de procesos en los que se puede ejecutar la función proporcionada.

El Listado 14.5 muestra una versión del servidor multiproceso que utiliza la clase `multiprocessing.Pool`. Sin embargo, este programa funciona de un modo diferente. A diferencia del servidor anterior (Listado 14.4) que crea y destruye un proceso por cada conexión, en este los procesos existen desde que se instancia la clase `Pool` y se reutilizan una y otra vez para las nuevas conexiones. Por eso a esta técnica se le llama *preforking* y la ventaja es evidente: la aplicación se ahorra la sobrecarga de crear y destruir procesos continuamente, y puede ser una diferencia significativa si el servidor recibe muchas peticiones que requieren poco procesamiento. También tiene una desventaja: el consumo de memoria es el mismo aunque el servidor esté completamente ocioso.

```
import sys
import time
import socket
from multiprocessing import Pool

MAX_PROCS = 10

def upper(msg):
    time.sleep(1) # simulates a complex job
    return msg.upper()

def handle(sock, client):
    print(f"Client connected: {client}")
    while 1:
        data = sock.recv(32)
        if not data:
            break
        sock.sendall(upper(data))
```

¹Este código pertenece al módulo `socketserver` de la librería estándar.

²No conozco ninguna traducción razonable para esto y me niego a llamarlo *piscina de procesos*, lo siento.

```

sock.close()
print(f"Client disconnected: {client}")

def main(port):
    sock = socket.socket()
    sock.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
    sock.bind('', port)
    sock.listen(5)

    with Pool(MAX_PROCS) as pool:
        while 1:
            conn, client = sock.accept()
            pool.apply_async(handle, (conn, client))

if len(sys.argv) != 2:
    print("Usage: {0} <port>".format(sys.argv[0]))
    sys.exit(1)

try:
    main(int(sys.argv[1]))
except KeyboardInterrupt:
    print("shut down")

```

LISTADO 14.5: Servidor TCP con *preforking*
 @/upper/tcp_prefork_pool.py

Hay otro modo más agresivo de implementar *preforking* con TCP que consiste en compartir el *master socket* entre los procesos hijos y todos ellos invocan `accept()`. Eso es posible cuando el SO ofrece soporte específico (en el caso de Linux). Lo puedes ver en el Listado 14.6.

```

import sys
import os
import time
import socket
from multiprocessing import Pool

MAX_PROCS = 10

def upper(msg):
    time.sleep(1) # simulates a complex job
    return msg.upper()

def handle(sock, client):
    print(f"Client connected: {client}, PID: {os.getpid()}")
    while 1:
        data = sock.recv(32)
        if not data:
            break
        sock.sendall(upper(data))

    sock.close()
    print(f"Client disconnected: {client}")

def worker(sock):

```

```

try:
    while 1:
        conn, client = sock.accept()
        handle(conn, client)
except KeyboardInterrupt:
    sys.exit(0)

def main(port):
    sock = socket.socket()
    sock.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
    sock.bind('', port)
    sock.listen(5)

    with Pool(MAX_PROCS) as pool:
        pool.map(worker, [sock] * MAX_PROCS)

if len(sys.argv) != 2:
    print("Usage: {0} <port>".format(sys.argv[0]))
    sys.exit(1)

try:
    main(int(sys.argv[1]))
except KeyboardInterrupt:
    print("shut down")

```

LISTADO 14.6: Servidor TCP con *preforking* para el método `accept()`
 `/upper/tcp_prefork_pool_accept.py`

14.7. Servidor UDP

Es momento de ver las variantes UDP del servicio *upper*. El Listado 14.7 es una implementación funcional y correcta del servidor salvo porque, como de costumbre en el libro, obviamos el tratamiento de errores en favor de la legibilidad³. Hay unas cuantas diferencias evidentes y alguna no tan evidente que vale la pena destacar respecto al servidor TCP.

En la función `main()` hay un bucle que recibe mensajes, e imitando la estructura del servidor TCP, invoca la función `handle()` con cada mensaje⁴ y el endpoint del cliente que lo ha enviado. La función `handle()` hace el trabajo y envía la respuesta. No hay un bucle en esa función, lo que deja patente que este es un servidor orientado a mensajes, no a conexiones.

Recuerda que con UDP la E/S parcial no es un problema. Si no hay pérdida de datos⁵, cada llamada a `recvfrom()` devuelve un mensaje completo y cada llamada a `sendto()` envía el mensaje completo.

³No olvides tenerlo en cuenta si escribes código para producción.

⁴No con cada conexión, que no hay.

⁵Y recuerda que eso es factible con UDP.

```

import sys
import time
import socket

def upper(msg):
    time.sleep(1) # simulates a complex job
    return msg.upper()

def handle(sock, msg, client, n):
    print(f"New request: {n} {client}")
    sock.sendto(upper(msg), client)

def main(port):
    sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
    sock.bind('', port)

    n = 0
    while 1:
        msg, client = sock.recvfrom(1024)
        handle(sock, msg, client, n := n+1)

if len(sys.argv) != 2:
    print("Usage: {0} <port>".format(sys.argv[0]))
    sys.exit(1)

try:
    main(int(sys.argv[1]))
except KeyboardInterrupt:
    print("shut down")

```

LISTADO 14.7: Servidor UDP

 /upper/udp_server.py

El cliente UDP lo puedes ver en el Listado 14.8. Como en TCP, puedes sustituirlo por ncat si quieres.

```

import sys
from socket import socket, SOCK_DGRAM

def main(host, port):
    with socket(type=SOCK_DGRAM) as sock:
        server_endpoint = (host, port)

        while 1:
            data = sys.stdin.readline().strip().encode()
            if not data:
                break

            sock.sendto(data, server_endpoint)
            msg, _ = sock.recvfrom(1024)
            print("Reply is '{}'.format(msg.decode())

if len(sys.argv) != 3:
    print("Usage: {0} <host> <port>".format(sys.argv[0]))
    sys.exit(1)

try:

```

```
main(sys.argv[1], int(sys.argv[2]))
except KeyboardInterrupt:
    print("shut down")
```

LISTADO 14.8: Cliente UDP
📄/upper/udp_client.py

También tenemos 📄/upper/udp_stress_client.py, el cliente de *stress* para UDP. Pruébalo con este servidor y con las variantes que vamos a ver a continuación para ver las diferencias. Aquí puedes ver cómo se comporta con el servidor UDP básico.

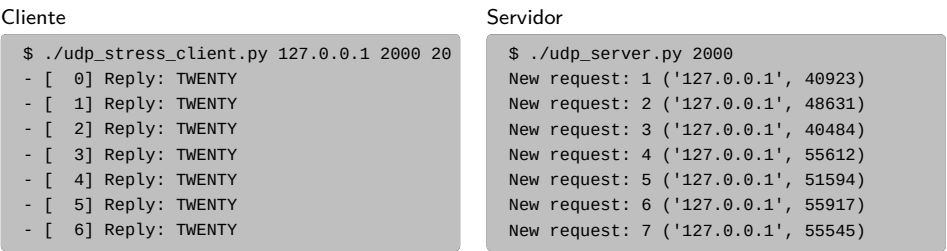


FIGURA 14.4: Servidor y cliente upper TCP

A la vista de esta salida se puede apreciar esa diferencia no tan evidente de la que hablábamos antes. Resulta que este no es un servidor iterativo, al menos no en el sentido en el que lo es el TCP. Cada llamada a `recvfrom()` puede recoger un mensaje de un cliente diferente, lo que eventualmente permite progresar a todos los clientes que estén enviando solicitudes. Por eso, este es un servidor **concurrente** (ver Figura 14.5). Es importante recordar aquí la diferencia entre *concurrente* y *paralelo*. Este servidor no es paralelo, no puede ejecutar la función `handle()` exactamente al mismo tiempo para varios clientes, pero eso no es necesario para considerarlo concurrente. La tabla 14.1 resume este aspecto para los servidores que hemos visto. Es una consecuencia directa del diferente modo de trabajar de TCP y UDP.

Servidor	Concurrente	Paralelo
TCP iterativo	No	No
TCP multihilo	Sí	Sí
TCP multiproceso	Sí	Sí
UDP básico	Sí	No

CUADRO 14.1: Concurrencia y paralelismo de los servidores TCP y UDP

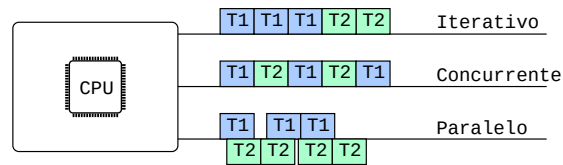


FIGURA 14.5: Comparación entre procesamiento iterativo, concurrente y paralelo

14.8. Servidor UDP multiproceso

Por supuesto es posible crear un servidor UDP que cree un proceso por cada mensaje recibido consiguiendo así un procesamiento paralelo. El Listado 14.9 muestra una posible implementación, en este caso utilizando la clase `multiprocessing.Process` en lugar de `fork()` directamente.

```
import sys
import time
import socket
import multiprocessing as mp

MAX_PROCS = 10

def start_new_process(func, args):
    for p in mp.active_children()[::-1][MAX_PROCS:]:
        p.join()

    ps = mp.Process(target=func, args=args)
    ps.start()

def upper(msg):
    time.sleep(1) # simulates a complex job
    return msg.upper()

def handle(sock, msg, client, n):
    print(f"New request: {n} {client}")
    sock.sendto(upper(msg), client)
    sys.exit(0)

def main(port):
    sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
    sock.bind(('', port))

    n = 0
    while 1:
        msg, client = sock.recvfrom(1024)
        start_new_process(handle, (sock, msg, client, n := n + 1))

if len(sys.argv) != 2:
    print("Usage: {0} <port>".format(sys.argv[0]))
    sys.exit(1)

try:
```



```

main(int(sys.argv[1]))
except KeyboardInterrupt:
    print("shut down")

```

LISTADO 14.9: Servidor UDP multiproceso
 /upper/udp_process.py

Sin embargo, si la tarea que realiza en cada petición es simple, la sobrecarga relativa por la creación y destrucción de procesos es mucho más grave que en el caso de TCP. Aquí se está creando un proceso por ¡cada mensaje!, no por cada conexión. Lo que sí puede ser razonable es utilizar la técnica de *preforking* que evita completamente esa sobrecarga. Lo puedes ver en el Listado 14.10.

```

import sys
import time
import socket
from multiprocessing import Pool

MAX_PROCS = 20

def upper(msg):
    time.sleep(1) # simulates a complex job
    return msg.upper()

def handle(sock, msg, client, n):
    print(f"New request: {n} {client}")
    sock.sendto(upper(msg), client)

def main(port):
    sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
    sock.bind(('', port))

    with Pool(MAX_PROCS) as pool:
        n = 0
        while 1:
            msg, client = sock.recvfrom(1024)
            pool.apply_async(handle, (sock, msg, client, n := n+1))

if len(sys.argv) != 2:
    print("Usage: {0} <port>".format(sys.argv[0]))
    sys.exit(1)

try:
    main(int(sys.argv[1]))
except KeyboardInterrupt:
    print("shut down")

```

LISTADO 14.10: Servidor UDP con *preforking*
 /upper/udp_prefork.py

14.9. Servidor UDP multihilo

Implementar un servidor UDP creando un hilo cada vez que llega un mensaje para ejecutar en él la función `handle()` no es una buena idea. La sobrecarga de crear hilos no es despreciable, aunque no tan grave ni de lejos como con procesos. Pero el mayor problema no es ese. El problema es que en UDP todos los hilos tienen acceso al mismo socket, a diferencia de servidor TCP multihilo, en el que cada hilo opera su propio socket. Compartir el socket provoca condiciones de carrera, bloqueos, etc. Por eso, un servidor UDP multihilo de este tipo no es nada recomendable.

Sin embargo, es posible aplicar un enfoque distinto. Utilizar el paralelismo que proporcionan los hilos solo a la tarea del servidor, sin involucrar a los sockets. La idea es tener un hilo para recibir las peticiones (el principal), otro para enviar las respuestas (*responder*) y un pool para el procesamiento de los mensajes. Los mensajes recibidos se pasan como argumento a la función `handle()`, que se ejecuta en el pool. Las respuestas se envían al hilo *responder* por medio de una cola (`queue.Queue`). De ese modo, no hay ningún socket compartido, tenemos las ventajas, pero no los inconvenientes. Lo puedes ver en el Listado 14.11.

```
import sys
import time
import queue
from socket import socket, SOCK_DGRAM
from threading import Thread, current_thread
from concurrent.futures import ThreadPoolExecutor

MAX_THREADS = 20
output_queue = queue.Queue()

def upper(msg):
    time.sleep(1) # Simulate heavy processing
    return msg.upper()

def handle(msg, client, n):
    print(f"Processing request {n} from {client}, thread {current_thread().name}")
    response = upper(msg)
    output_queue.put((response, client))

def responder(sock):
    while 1:
        response, client = output_queue.get()
        sock.sendto(response, client)

def main(port):
    sock = socket(type=SOCK_DGRAM)
    sock.bind('', port)

    Thread(target=responder, args=(sock,), daemon=True).start()
```

```

with ThreadPoolExecutor(max_workers=MAX_THREADS) as executor:
    n = 0
    while 1:
        msg, client = sock.recvfrom(1024)
        executor.submit(handle, msg, client, n := n+1)

if __name__ == '__main__':
    if len(sys.argv) != 2:
        print("Usage: {0} <port>".format(sys.argv[0]))
        sys.exit(1)

    try:
        main(int(sys.argv[1]))
    except KeyboardInterrupt:
        print("shut down")
    
```

LISTADO 14.11: Servidor UDP con pool de hilos

📄/upper/udp_threadpool.py

14.10. socketserver

El módulo `socketserver` de la librería estándar de Python ofrece clases y *mixins* para implementar servidores TCP y UDP de un modo muy sencillo. El Listado 14.12 es la versión equivalente al servidor TCP iterativo.

```

import sys
import os
import time
from socketserver import StreamRequestHandler, TCPServer

def upper(msg):
    time.sleep(1) # simulates a complex job
    return msg.upper()

class Handler(StreamRequestHandler):
    def handle(self):
        print(f"Client connected: {self.client_address}")
        while 1:
            data = os.read(self.rfile.fileno(), 32)
            if not data:
                break

            self.wfile.write(upper(data))
            print(f"Client disconnected: {self.client_address}")

class CustomTCPServer(TCPServer):
    allow_reuse_address = True

if len(sys.argv) != 2:
    print("Usage: {0} <port>".format(sys.argv[0]))
    sys.exit(1)

server = CustomTCPServer(('', int(sys.argv[1])), Handler)
server.serve_forever()
    
```

LISTADO 14.12: Servidor TCP iterativo con socketserver
📄/upper/tcp_ss.py

La clase `TCPServer` permite crear un servidor TCP iterativo. Solo hay que pasarle como argumento el endpoint donde debe escuchar y una clase que herede de `StreamHandlerRequest` que proporcione a su vez un método `handle()` que es esencialmente equivalente a la función del mismo nombre de los ejemplos anteriores. Una diferencia muy evidente es que esa función no invoca `send()` y `recv()` sobre el socket conectado. En su lugar, invoca `read()` y `write()` sobre los *file objects* llamados `rfile` y `wfile`, pero que obviamente se refieren al socket.

Otra peculiaridad es que no se usa directamente la clase `TCPServer`. El programa define una clase `CustomTCPServer` que hereda de ella para redefinir el atributo `allow_reuse_address` como `True`. Eso es equivalente a la activación de la opción `SO_REUSEADDR`.

El módulo `socketserver` proporciona las clases `ForkingTCPServer` y `ThreadingTCPServer` alternativas a `TCPServer` para implementar directamente servidores multiproceso y multihilo, respectivamente. Y efectivamente, basta con cambiar la clase como puedes comprobar en 📄/upper/tcp_ss_thread.py, que sería equivalente al Listado 14.3 y 📄/upper/tcp_ss_fork.py que sería equivalente al Listado 14.4.

También proporciona la clase `UDPServer`, que puedes ver en uso en el Listado 14.13 y es el equivalente al servidor del Listado 14.7. Aquí tiene como manejador una clase que hereda de `DatagramRequestHandler` y especializa el método `handle()` que recibe una petición y envía la respuesta.

```
import sys
import time
from socketserver import DatagramRequestHandler, UDPServer

def upper(msg):
    time.sleep(1) # simulates a complex job
    return msg.upper()

class Handler(DatagramRequestHandler):
    def handle(self):
        print(f"New request: {self.client_address}")
        msg = self.rfile.read()
        self.wfile.write(upper(msg))

if len(sys.argv) != 2:
    print("Usage: {0} <port>".format(sys.argv[0]))
    sys.exit(1)

server = UDPServer(('', int(sys.argv[1])), Handler)
server.serve_forever()
```

LISTADO 14.13: Servidor UDP con `socketserver`
📄/upper/udp_ss.py

También existen las clases `ForkingUDPServer` y `ThreadingUDPServer` para implementar servidores multiproceso y multihilo, respectivamente, pero que tienen los inconvenientes que ya hemos expuesto.

El módulo no tiene soporte para *preforking* en ninguna de las dos modalidades y tampoco son sencillas de implementar porque pasar las instancias de las clases involucradas a los subprocesos es complejo.

14.11. Operaciones bloqueantes

En §6.8 vimos algunas de las consecuencias de que los sockets al igual que los archivos requieran de operaciones de E/S. Pero hay otra consecuencia muy importante que no habíamos abordado aún: Estas operaciones son **bloqueantes**. Igual que leer de la consola esperando que el usuario pulse una tecla puede conllevar que el proceso quede bloqueado indefinidamente, lo mismo ocurre con el método `recv()`. En estos casos, el SO bloquea el proceso que invoca estas funciones hasta que lleguen los datos esperados. Pero no es algo que afecte solo a `recv()`; también afecta a `accept()` y por supuesto a `recvfrom()`, pues también en esos casos el proceso espera a la llegada de mensajes procedentes de la red para continuar su ejecución.

Es bastante intuitivo pensar que estas operaciones de «lectura» sean bloqueantes, pero resulta que también lo son las operaciones de «escritura» como `send()`, `sendto()` o `connect()`. Aunque es menos habitual que el envío de un mensaje conlleve el bloqueo de forma indefinida⁶, también son operaciones de E/S y por eso el SO puede bloquear el proceso a la espera de que la operación se complete; en este caso, que los datos sean enviados, o almacenados en el buffer correspondiente, antes de continuar la ejecución del programa.

Que estas operaciones sean bloqueantes puede sonar como un problema que hay que resolver, pero es lógico que sea así e incluso resulta conveniente. Por norma general es **más sencillo** escribir —y más eficiente ejecutar— un programa de comunicaciones, o en general intensivo en E/S, si se comporta de este modo.

14.12. Alternativas a la E/S bloqueante

Pese a lo dicho, hay algunas situaciones en las que el programa podría necesitar aprovechar los tiempos «muertos» mientras el SO espera a que

⁶Volveremos sobre esto cuando hablemos de control de flujo en el capítulo 11

estas operaciones terminen, o quizá quiera ejecutar otras operaciones de E/S. Recuerda que cuando un proceso está bloqueado, la CPU no está ociosa; el planificador del SO pone otro proceso en ejecución, pero ciertamente, tu aplicación está parada.

Normalmente hay tres opciones para aprovechar los bloqueos: operaciones no bloqueantes, timeouts y programación asíncrona. Aunque esto es potencialmente aplicable a cualquier operación de E/S, lógicamente aquí vamos a hablar de sockets. Veámoslas brevemente:

Operaciones no bloqueantes

Los sockets están por defecto en modo bloqueante y por eso se comportan como hemos visto. Se puede conseguir que un socket concreto tenga un comportamiento no bloqueante invocando su método `setblocking(False)`. A partir de ese momento, si por ejemplo se invoca `recv()` y no hay datos disponibles, eleva inmediatamente una excepción `BlockingIOError` y es responsabilidad del programador capturar la excepción e intentar la recepción de nuevo más tarde si así lo estima oportuno. No es difícil imaginar cómo esto nos puede complicar rápidamente las cosas... Además es necesario señalar que el comportamiento de los sockets no bloqueantes puede diferir entre plataformas, algo que dificulta la escritura de código portable, otra buena razón para evitarlos si no está realmente justificado.

Timeouts

Hay otras situaciones en las que las operaciones bloqueantes son adecuadas, pero no es aceptable que el proceso quede bloqueado indefinidamente. En esta situación se puede usar un timeout, es decir, definir un tiempo máximo que de alcanzarse provoca la interrupción de la operación y eleva una excepción `TimeoutError` que avisa de la situación devolviendo así el control al programa. El método `socket.settimeout()` permite fijar ese límite (expresado en segundos).

Este método acepta dos valores especiales. Si se llama con `None`, elimina el timeout y el socket queda en modo bloqueante, pero si se llama con `0`, el socket cambia a modo no bloqueante, es decir, es como invocar `setblocking(False)`.

Entrada/salida asíncrona

El SO ofrece algunas llamadas al sistema para gestionar operaciones de E/S de forma asíncrona. La más básica y rudimentaria, que veremos a continuación, es `select()`, que permite esperar a que uno o más descriptores

de archivo—en el sentido amplio del concepto— estén listos para leer o escribir.

Aparte de eso, Python proporciona un modelo de programación asíncrona basado en corutinas que funcionan en un único hilo de ejecución. La programación asíncrona es potente y compleja, y requiere del programador un enfoque muy diferente a la programación secuencial habitual.

Veremos una pequeña introducción a estas dos soluciones en las siguientes secciones.

14.13. Servidor TCP asíncrono con `select()`

Con `select()` se puede resolver de otro modo el problema del servidor iterativo: los dos puntos de bloqueo en `accept()` y `recv()`.

La función `select()`, que en Python se encuentra en el módulo del mismo nombre, permite manejar una serie de descriptors de archivo cuyo estado puede cambiar en cualquier momento. El prototipo de la función es este:

```
read_ready, write_ready, error_ready = select.select(rlist, wlist, xlist, timeout=None)
```

Los parámetros `rlist`, `wlist` y `xlist` son listas de descriptors de archivo que quieres que `select()` monitorice. En la lista `rlist` pones descriptors de los que quieres leer, en `wlist` de los que quieres escribir y en `xlist` de los que quieres comprobar si han tenido algún error. En los sistemas POSIX los descriptors de archivo incluyen muchas otras cosas como tuberías, dispositivos, terminales, y por supuesto, también sockets.

El valor de retorno es una tupla con tres listas, que contienen los descriptors que efectivamente están listos para ser leídos, escritos o han tenido un error, es decir, subconjuntos de los argumentos de la función.

La función `select()` bloquea el proceso hasta que ocurre algo en algún descriptor de cualquiera de las listas. Eso proporciona un único punto de bloqueo en el programa y por tanto, permite manejar múltiples entradas asíncronas simultáneas sin necesidad de hilos u otros mecanismos para conseguir paralelismo.

Puedes ver el código del servidor completo en el Listado 14.14. Como hay dos tipos de sockets: el máster y los conectados a los clientes, el programa debe hacer un tratamiento diferente para cada uno, que corresponde con los métodos `master_handler()` y `child_handler()` respectivamente. La lista `socks` contiene inicialmente el socket maestro, y conforme el servidor acepta conexiones, los nuevos sockets se añaden a esa misma lista. La lista `socks`

se pasa como primer argumento a `select()` porque en este programa tan sencillo solo nos interesa evitar el bloqueo relacionado con las lecturas.

```
import sys
import socket
import time
import select
from utils import show_select_status

def upper(msg):
    time.sleep(1) # simulates a complex job
    return msg.upper()

class Server:
    def __init__(self, port):
        self.master = socket.socket()
        self.master.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
        self.master.bind('', port)
        self.master.listen(5)
        self.socks = [self.master]

    def master_handler(self):
        conn, client = self.master.accept()
        self.socks.append(conn)
        print(f"- Client connected: {client}, Total {len(self.socks)} sockets")

    def child_handler(self, conn):
        data = conn.recv(32)
        if not data:
            self.socks.remove(conn)
            conn.close()
            return

        conn.sendall(upper(data))

    def run(self):
        while 1:
            read_ready = select.select(self.socks, [], [])[0]
            show_select_status(self.socks, read_ready)
            for s in read_ready:
                if s == self.master:
                    self.master_handler()
                else:
                    self.child_handler(s)

if __name__ == "__main__":
    if len(sys.argv) != 2:
        print("Usage: {0} <port>".format(sys.argv[0]))
        exit(1)

    try:
        server = Server(int(sys.argv[1]))
        server.run()
    except KeyboardInterrupt:
        print("\nServer shutting down.")
        exit(0)
```


LISTADO 14.14: Servidor TCP con `select()``📄/upper/tcp_select.py`

Si pruebas este servidor con el cliente de stress podrás ver que su comportamiento es, salvando las diferencias, similar al del servidor UDP básico. Ofrece concurrencia: todos los clientes progresan, pero no paralelismo: la función `upper()` no se ejecuta a la vez para varios clientes.

```
$ ./tcp_stress_client.py 127.0.0.1 2000 4
- [ 0] Reply: TWENTY
- [ 1] Reply: TWENTY
- [ 2] Reply: TWENTY
- [ 3] Reply: TWENTY
- [ 0] Reply: TINY
```

La librería estándar de Python incluye un módulo llamado `selectors` con un nivel de abstracción algo más alto para conseguir lo mismo. Puedes ver y probar un servidor para el servicio upper con `selectors` en el archivo `📄/upper/tcp_selectors.py`. Se basa en la creación de un objeto de clase `DefaultSelector` en el que se registran manejadores para un descriptor de archivo y un motivo específico.

```
selector = selectors.DefaultSelector()
selector.register(f, selectors.EVENT_READ, handler)
```

Esto le indica al `selector` que si hay datos disponibles para leer en el descriptor de fichero `f`, debe invocar la función `handler`.

14.14. Servidor TCP asíncrono con `asyncio`

Python, como muchos otros lenguajes modernos, ofrece una solución más general: concurrencia basada en *corutinas* o tareas cooperativas. Con este modelo, el código se organiza de un modo similar a un programa secuencial convencional, pero cuando se invoca una primitiva de E/S bloqueante, el proceso no se bloquea, sino que suspende la ejecución de esa función y cede el control a otra. De ese modo, el proceso puede aprovechar los tiempos «muertos» sin bloqueos y sin necesidad de crear procesos u hilos adicionales.

El programador debe indicar explícitamente qué funciones son corutinas con la palabra clave `async`, y debe llamarlas con `await`. Las corutinas pueden a su vez invocar otras corutinas, mientras que la función más externa la debe gestionar un bucle de eventos. Este bucle es el que se encarga de ejecutar las corutinas, suspenderlas y reanudarlas cuando corresponda. El bucle de eventos y otras muchas utilerías las ofrece el módulo `asyncio`, que literalmente es la abreviatura de *asynchronous I/O*.

Existen dos modalidades para implementar un servidor TCP con `asyncio`:

- *Streams*
- *Transports and Protocols*

Streams es la de más alto nivel y cómo su nombre indica se basa en el manejo de dos flujos: uno de lectura y otro de escritura. El Listado 14.15 aplica ese enfoque. Se proporciona una función —que aquí hemos llamado `handle()`— que recibe los flujos (`reader` y `writer`). Para crear el servidor se utiliza `asyncio.start_server()`. Fíjate que todas las funciones bloqueantes (o las que las llaman) son asíncronas (declaradas `async`) y son invocadas con `await`. Aunque se parece bastante al servidor TCP iterativo, este puede intercalar la recepción de mensajes, que se hace en `reader.read`, con la aceptación de nuevas conexiones.

Con `async` la función que simula un `upper()` costoso no puede ser una llamada a `sleep()` porque el bucle de eventos utilizaría esa pausa para ejecutar otras corutinas, de modo que no conseguiría su propósito de simular una tarea costosa. En su lugar se ha añadido un bucle vacío que dura 1 segundo —función `fake_heavy_upper()`.

```
import sys
import asyncio
import time

async def upper(msg):
    def fake_heavy_upper(msg):
        start = time.time()
        while time.time() - start < 1:
            pass
        return msg.upper()

    return await asyncio.to_thread(fake_heavy_upper, msg)

async def handle(reader, writer):
    peername = writer.get_extra_info('peername')
    print(f"Client connected: {peername}")

    try:
        while 1:
            data = await reader.read(32)
            if not data:
                break
            writer.write(await upper(data))
            await writer.drain()

    except asyncio.CancelledError:
        pass
    finally:
        print(f"Client disconnected: {peername}")
        writer.close()
```

```

        await writer.wait_closed()

    async def main(port):
        server = await asyncio.start_server(handle, '', port)

        async with server:
            await server.serve_forever()

    if len(sys.argv) != 2:
        print("Usage: {0} <port>".format(sys.argv[0]))
        sys.exit(1)

    try:
        asyncio.run(main(sys.argv[1]))
    except KeyboardInterrupt:
        print("shut down")

```

LISTADO 14.15: Servidor TCP con `asyncio` Streams`📄/upper/tcp_async_streams.py`

La segunda modalidad para implementar un servidor es *Transports and Protocols*, que es de más bajo nivel que *Streams*. El *transporte* representa el medio para transmitir los mensajes (el socket), mientras que el *protocolo* dice qué información hay que transmitir y cómo se debe interpretar.

En el caso más sencillo consiste en implementar una clase que hereda de `asyncio.Protocol` y encapsula el comportamiento del protocolo de aplicación. Se consigue especializando los métodos de la clase `Protocol`: `connection_made()`, `data_received()`, etc., que son invocados cuando ocurren esos eventos. A esto se le llama programación basada en *hooks*. El Listado 14.16 muestra el servidor TCP con esta modalidad.

```

import sys
import asyncio
import time

    async def upper(msg):
        def fake_heavy_upper(msg):
            start = time.time()
            while time.time() - start < 1:
                pass
            return msg.upper()

        return await asyncio.to_thread(fake_heavy_upper, msg)

    class UpperProtocol(asyncio.Protocol):
        def connection_made(self, transport):
            self.peername = transport.get_extra_info("peername")
            print(f"Client connected: {self.peername}")
            self.transport = transport

        def data_received(self, data):
            message = data.decode()
            loop = asyncio.get_running_loop()

```

```

        loop.create_task(self.handle_message(message))

    async def handle_message(self, message):
        reply = await upper(message)
        self.transport.write(reply.encode())

    def connection_lost(self, exc):
        print(f"Client disconnected: {self.peername}")
        self.transport.close()

    async def main(port):
        loop = asyncio.get_running_loop()
        server = await loop.create_server(UpperProtocol, '', port)

        async with server:
            await server.serve_forever()

if len(sys.argv) != 2:
    print("Usage: {0} <port>".format(sys.argv[0]))
    sys.exit(1)

try:
    asyncio.run(main(int(sys.argv[1])))
except KeyboardInterrupt:
    print('shut down')
```

LISTADO 14.16: Servidor TCP con asyncio Transport and Protocols
/upper/tcp_async_protocol.py

14.15. Cliente TCP asíncrono con asyncio

Una posible versión asíncrona del cliente TCP que hemos visto en 14.3 podría ser el del Listado 14.17. Es destacable el uso del método `run_in_executor()` que permite ejecutar la función `stdin.readline()`, que es bloqueante, en un hilo independiente para evitar que bloquee el bucle de eventos. En realidad aunque el envío y la recepción son concurrentes, no aporta ninguna ventaja en este caso y no tiene demasiado interés práctico. Evitar el bloqueo de la lectura desde la consola y desde el socket no va a mejorar el rendimiento del cliente, seguirá pudiendo enviar un mensaje cada segundo como máximo.

```

import sys
import asyncio

def readline():
    return sys.stdin.readline().strip().encode()

    async def main(host, port):
        reader, writer = await asyncio.open_connection(host, port)

        try:
            while True:
```

```

data = await asyncio.get_event_loop().run_in_executor(None, readline)
if not data:
    break

writer.write(data)
await writer.drain()

msg = b''
while len(msg) < len(data):
    chunk = await reader.read(32)
    if not chunk:
        break
    msg += chunk

print(f"Reply is '{msg.decode()}'")

except asyncio.CancelledError:
    pass
finally:
    writer.close()
    await writer.wait_closed()

if len(sys.argv) != 3:
    print("Usage: {0} <host> <port>".format(sys.argv[0]))
    sys.exit(1)

try:
    asyncio.run(main(sys.argv[1], int(sys.argv[2])))
except KeyboardInterrupt:
    print("shut down")

```

LISTADO 14.17: Cliente TCP con `asyncio`
 📄/upper/tcp_client_async.py

Eso no significa que no sea interesante en otros casos. Si el cliente tiene otras cosas que hacer, puede suponer una importante ventaja. Un ejemplo práctico de esto es el cliente de *stress* que hemos estado usando a lo largo del capítulo: 📄/upper/tcp_stress_client.py. Este cliente envía mensajes a varios servidores, de modo que mientras espera la respuesta de uno, puede estar enviando mensajes a otros. Te recomendamos estudiar ese programa y compararlo con el cliente 14.17 que acabamos de ver.

14.16. Servidor UDP asíncrono con `asyncio`

También puedes implementar un servidor UDP con la modalidad *Transports and Protocols* (ver Listado 14.18), heredando de `asyncio.DatagramProtocol`. Sin embargo, no es posible implementar un servidor UDP con *Streams* porque como vimos antes, está diseñado solo para transferencia de datos orientada a flujo.

```
import sys
import asyncio
import time

async def upper(msg):
    def fake_heavy_upper(msg):
        start = time.time()
        while time.time() - start < 1:
            pass
        return msg.upper()

    return await asyncio.to_thread(fake_heavy_upper, msg)

class UpperUDPProtocol(asyncio.DatagramProtocol):
    def connection_made(self, transport):
        self.transport = transport

    def datagram_received(self, data, addr):
        print(f"New request: {addr}")
        asyncio.create_task(self.handle_request(data, addr))

    async def handle_request(self, data, addr):
        response = await upper(data.decode())
        self.transport.sendto(response.encode(), addr)

    def error_received(self, exc):
        print(f"Comm error: {exc}")

async def main(port):
    loop = asyncio.get_running_loop()

    transport, _ = await loop.create_datagram_endpoint(
        lambda: UpperUDPProtocol(),
        local_addr=(' ', port)
    )

    try:
        await asyncio.sleep(3600)
    except asyncio.CancelledError:
        pass
    finally:
        transport.close()

if len(sys.argv) != 2:
    print("Usage: {0} <port>".format(sys.argv[0]))
    sys.exit(1)

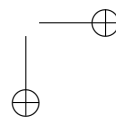
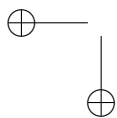
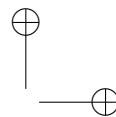
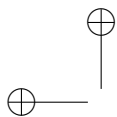
try:
    asyncio.run(main(int(sys.argv[1])))
except KeyboardInterrupt:
    print("shut down")
```

LISTADO 14.18: Servidor UDP con *asyncio Transport and Protocols*
📄/upper/udp_async_protocol.py

Y ¿qué más?

Hemos visto las posibilidades *nativas* de concurrencia que ofrece un servidor UDP debido su naturaleza orientada a mensajes individuales. Al mismo tiempo eso implica limitaciones debido a que es necesario realizar todas sus operaciones sobre un único socket, por la contención que supone proteger el socket adecuadamente. A pesar de ello, hemos visto técnicas como el *pool* de hilos o el *preforking* que permiten un alto nivel de paralelismo evitando el acceso compartido. Por contra, TCP ofrece más opciones para hacer servidores de alto rendimiento al estar basado en un modelo de conexiones que permite que cada una de esas conexiones disponga de su propio socket, con el aislamiento que ello conlleva. Sea como sea, no se deben despreciar las posibilidades de concurrencia que ofrece la gestión de E/S asíncrona, ya sea con `select` y sus derivados, o el soporte integrado en el lenguaje gracias a `asyncio`.

Aunque solo hemos visto una pincelada de cada una de estas técnicas, es sencillo imaginar las diversas opciones disponibles, tanto con Python como con otros lenguajes. Te invitamos a experimentar por ti mismo y aplicar estas y otras técnicas más avanzadas a tus propios proyectos.



Capítulo 15

Publicador-Suscriptor

Al terminar este capítulo, entenderás:

- Qué es el modelo de interacción publicador-suscriptor.
- Cómo implementar un chat con este modelo de interacción, tanto con UDP como con TCP.
- Qué es y qué ofrece el protocolo MQTT para implementar este tipo de comunicación.

El segundo modelo de interacción más habitual, por detrás del cliente-servidor, es publicador-suscriptor¹. En lugar de aplicar una interacción basada en petición-respuesta, este modelo asume existen dos tipos de participantes: los *publicadores* envían mensajes a un espacio común (el *canal*) y los *suscriptores* reciben los mensajes que les interesan en función de a qué estén suscritos. El matiz clave es que esos mensajes se envían sin esperar una respuesta. Si los publicadores esperan algún tipo de información de vuelta, lo harán también con un rol de suscriptores, pero no como parte de la misma interacción, por lo que no podemos considerar que se trate de respuestas.

El modelo se caracteriza por el uso de un componente intermedio llamado **broker** que es quien se encarga de recibir los mensajes que proceden de los publicadores y hacerlos llegar a los suscriptores. Lógicamente, no tiene sentido llamar *peticiones* a estos mensajes porque no buscan un resultado. En su lugar se les llama *publicaciones* o más comúnmente *eventos*.

El hecho de que haya mensajes solo en un sentido y que todos ellos pasen por el *broker* desacopla completamente a publicadores y suscriptores. Lo único que deben conocer todos los participantes es el endpoint del broker.

¹Abreviado a menudo como «pubsub».

La otra peculiaridad de este modelo es que puede haber múltiples suscriptores para un mismo canal. El broker es el encargado de realizar y distribuir las copias necesarias para entregar a cada suscriptor..

15.1. Chat UDP para dos

Para entender cómo funciona este modelo, vamos a desarrollar paso a paso un ejemplo práctico típico: un *chat*. Es un programa sencillo que permite a varias personas enviar mensajes a una *sala* de modo que todos los usuarios presentes en la sala pueden verlos.

Empezaremos desde una versión extremadamente simple con solo dos participantes. Después iremos añadiendo funcionalidad y resolviendo los problemas que vayan apareciendo. Para facilitar el desarrollo inicialmente le vamos a dar la estructura cliente-servidor, pero conforme evolucione, incorporaremos un broker y se convertirá en publicación-suscripción. También por simplicidad, empezaremos con una versión UDP.

15.1.1. Paso 1: Mensaje unidireccional

En este primer paso los objetivos son crear:

- Un servidor que recibe un único mensaje, lo imprime en consola y termina.
- Un cliente que envía la cadena `hello` al servidor y termina.

Servidor

Las tareas que realiza el servidor son muy simples:

1. Crea un socket UDP.
2. Vincula el socket a un puerto libre.
3. Espera un datagrama que contiene un mensaje de texto.
4. Imprime el mensaje en consola.

Estas tareas corresponden línea a línea con el Listado 15.1.

```
import socket

sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
sock.bind(('', 12345))
message, client = sock.recvfrom(1024)
print(message.decode(), client)
sock.close()
```

LISTADO 15.1: Servidor de chat UDP básico
📄/chat-udp/server1.py

No hay mucho que comentar. Es un programa aún más simple de los que ya habíamos visto. Recuerda que al ser UDP, el argumento de `recvfrom()` limita el tamaño de los mensajes que podrán intercambiar los usuarios, pero para un chat sencillo como este, 1024 bytes parece más que suficiente.

Para ejecutarlo simplemente escribe el siguiente comando en la consola:

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

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

Cliente

Antes de escribir el programa cliente en Python es buena idea probar que el servidor funciona como debe. Y para eso puedes utilizar `ncat` (ver 6.12).

Con el siguiente comando puedes ejecutar un cliente UDP que envía la cadena `hola` al servidor que está escuchando en el puerto 12345 (el de acabas de poner en marcha). En este instante el servidor debería imprimir el saludo y terminar.

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

Ahora que has comprobado que el servidor funciona, puedes pasar a escribir un cliente que imite esta misma funcionalidad. También es muy simple: crea el socket UDP, envía un mensaje y termina. Lo puedes ver en el Listado 15.2.

```
import socket

sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
sock.sendto("hola".encode(), ('127.0.0.1', 12345))
sock.close()
```

LISTADO 15.2: Cliente de chat UDP básico
📄/chat-udp/client1.py

Como puedes comprobar por el segundo parámetro de la función `sendto()`, el cliente envía el mensaje al endpoint `127.0.0.1:12345`, es decir, estamos asumiendo que vas a ejecutar el servidor y el cliente en la misma máquina, algo recomendable al menos mientras escribes y pruebas el programa.

Lo puedes ejecutar como sigue, y el efecto debería ser el mismo que con `ncat`.

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

15.1.2. Paso 2: Devuelve el saludo

El servidor del paso anterior sólo imprime el mensaje recibido. Con un pequeño cambio podrá 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 vuelta (Listado 15.3).

```
import socket

sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
sock.bind(('', 12345))
message, peer = sock.recvfrom(1024)
print(message.decode(), peer)
sock.sendto("qué tal?".encode(), peer)
sock.close()
```

LISTADO 15.3: Servidor de chat UDP con respuesta
 chat-udp/server2.py

El cliente del paso anterior terminaba inmediatamente después de enviar su mensaje. Ahora debe esperar para recibir el mensaje del servidor. Como es lógico, basta con imitar lo que hace el servidor: usar el método `recvfrom()` (Listado 15.4).

```
import socket

sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
sock.sendto("hola".encode(), ('127.0.0.1', 12345))
message, peer = sock.recvfrom(1024)
print("{} from {}".format(message.decode(), peer))
sock.close()
```

LISTADO 15.4: Cliente de chat UDP con respuesta
 chat-udp/client2.py

15.1.3. Paso 3: Libertad de expresión

Con este paso los usuarios que ejecutan cliente y servidor tendrán realmente la posibilidad de conversar. En lugar de enviar una cadena literal, ambos programas van a leer de consola lo que el usuario teclee y lo van a enviar hacia el interlocutor. La conversación se mantendrá hasta que cualquiera de ellos envíe la cadena `bye`. El Listado 15.5 muestra el código completo del servidor después de este cambio:

```
import socket
QUIT = b"bye"

sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
```

```
sock.bind('', 12345))

while 1:
    message_in, peer = sock.recvfrom(1024)
    print(message_in.decode())

    if message_in == QUIT:
        break

    message_out = input().encode()
    sock.sendto(message_out, peer)

    if message_out == QUIT:
        break

sock.close()
```

LISTADO 15.5: Servidor de chat UDP por turnos

 /chat-udp/server3.py

La diferencia principal respecto a las versiones anteriores es el bucle `while`. Este bucle termina tanto si el usuario local introduce la cadena `bye` como si 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 15.6) es muy similar.

```
import socket
QUIT = b"bye"

sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
server = ('', 12345)

while 1:
    message_out = input().encode()
    sock.sendto(message_out, server)

    if message_out == QUIT:
        break

    message_in, peer = sock.recvfrom(1024)
    print(message_in.decode())

    if message_in == QUIT:
        break
```

LISTADO 15.6: Cliente de chat UDP por turnos

 /chat-udp/client3.py

Únicamente hay dos diferencias entre cliente y servidor: sólo el servidor ejecuta `bind()` y es el que empieza recibiendo un mensaje en el socket, mientras el cliente empieza leyendo de teclado. En UDP, el cliente tiene que ser el primero en enviar un mensaje. Como no hay conexión previa, el

servidor no puede saber quién es el cliente (su endpoint) hasta que reciba algo.

Esta versión tiene un problema grave en cuanto a la interacción entre los usuarios. 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. Es el mismo problema que ya vimos en los servidores del capítulo 14: la imposibilidad de atender múltiples entradas asíncronas simultáneamente². Pero en este caso concreto no tiene que ver con concurrencia. Aquí implica que los usuarios han de esperar a que su interlocutor envíe un mensaje antes de poder escribir de nuevo. O dicho de otro modo, los usuarios no pueden enviar 2 o más mensajes consecutivos, que es lo que se espera de un chat.

Y con esto además puedes ver que la situación no encaja con un protocolo petición-respuesta. No queremos que haya una correspondencia uno a uno entre los mensajes que envía el cliente y los que envía el servidor. Los mensajes que envía el servidor no son respuestas a solicitudes del cliente, cada usuario debe poder enviar tantos mensajes como quiera en cualquier momento.

15.1.4. Paso 4: Habla cuando quieras

Para resolver el problema de la E/S asíncrona vamos a utilizar hilos, que es la más sencilla al menos a nivel de código. Atenderemos la entrada procedente de la consola en un hilo adicional (el envío), mientras que el hilo principal se encargará de atender el socket (la recepción). El código del servidor aparece en el Listado 15.7.

```
4 import socket
5 import _thread
6 server = ('', 12345)
7 QUIT = b"bye"
8
9 class Chat:
10     def __init__(self, sock, peer):
11         self.sock = sock
12         self.peer = peer
13
14     def run(self):
15         _thread.start_new_thread(self.sending, ())
16         self.receiving()
17         self.sock.close()
18
19     def sending(self):
20         while 1:
21             message = input().encode()
```

²a menos que se utilicen hilos, procesos o E/S asíncrona.

```

22         self.sock.sendto(message, self.peer)
23
24         if message == QUIT:
25             break
26
27     def receiving(self):
28         while 1:
29             message, peer = self.sock.recvfrom(1024)
30             print(message.decode())
31
32             if message == QUIT:
33                 self.sock.sendto(QUIT, self.peer)
34                 break
35
36 if __name__ == '__main__':
37     sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
38     sock.bind(server)
39     message, client = sock.recvfrom(0, socket.MSG_PEEK)
40     Chat(sock, client).run()

```

LISTADO 15.7: Servidor de chat UDP simultaneo
 chat-udp/server4.py

El programa está compuesto por una clase `Chat` con tres métodos, aparte del 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 método `run()` crea y arranca el hilo para la tarea de envío y ejecuta la tarea de recepción.

En cuanto a la función principal, la llamada a `recvfrom()` (**línea 37**) se utiliza únicamente para obtener el endpoint del cliente, pero sin consumir nada del buffer del socket gracias al flag `MSG_PEEK`. La **línea 38** crea la instancia de la clase `Chat` pasando el socket y el endpoint del cliente como parámetros.

El cliente solo difiere en la creación del socket de modo que se puede reutilizar la clase `Chat` del servidor tal cual. Simplemente crea el socket y la instancia de `Chat`, a la que le pasa el socket y el endpoint del servidor. Puedes verlo en el Listado 15.8.

```

import socket
from server4 import Chat, server

sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
Chat(sock, server).run()

```

LISTADO 15.8: Cliente de chat UDP simultáneo
 chat-udp/client4.py

Sigue habiendo un pequeño problema estético. 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 habría que hacer que las tareas de recepción y envío escribieran en partes diferentes de la pantalla, o quizá construir un pequeño GUI. Sin embargo, ésta es una cuestión al margen de la finalidad de este ejemplo.

15.1.5. 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 es posible 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 15.9.

```
import sys
import socket
from threading import Thread

SERVER = ('', 12345)
QUIT = b'bye'

class Chat:
    def __init__(self, sock, peer):
        self.sock = sock
        self.peer = peer

    def run(self):
        sender_thread = Thread(target=self.sending, daemon=True)
        sender_thread.start()
        self.receiving()
        sender_thread.join()
        self.sock.close()

    def sending(self):
        while True:
            message = input().encode()
            self.sock.sendto(message, self.peer)
            if message == QUIT:
                break

    def receiving(self):
        while 1:
            message, _ = self.sock.recvfrom(1024)
            print("other> {}".format(message.decode()))
            if message == QUIT:
                self.sock.sendto(QUIT, self.peer)
                break

if __name__ == '__main__':
```



```

if len(sys.argv) != 2:
    print("usage: %s [--server|--client]" % sys.argv[0])
    sys.exit()

mode = sys.argv[1]
sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)

if mode == '--server':
    sock.bind(SERVER)
    message, client = sock.recvfrom(0, socket.MSG_PEEK)
    Chat(sock, client).run()
else:
    Chat(sock, SERVER).run()

```

LISTADO 15.9: Chat UDP multihilo (servidor y cliente)
 /chat-udp/chat-thread.py

15.2. Chat asíncrono con `select()`

Como alternativa a los hilos, veamos como implementar el chat anterior con `select()`. Como hemos visto, las dos entradas asíncronas a vigilar son la consola y el socket, así que le podemos pasar esos descriptores a `select()`. Es un uso más simple que el del servidor TCP de §14.13, pero que sigue la misma idea: proporcionar un único punto de bloqueo que permite atender cualquiera de las dos situaciones. La llamada sería algo como:

```

while 1:
    ready = select([sys.stdin, sock], [], [])[0]
    for fd in ready:
        if fd == sock:
            # leer del socket
        else:
            # leer de consola

```

El bucle `for` es necesario porque ambos descriptores podrían estar listos para leer al mismo tiempo, o muy próximos en el tiempo. Para leer se va a seguir usando las funciones `recvfrom()` e `input()`. Recuerda que, después de que `select()` haya confirmado que hay datos disponibles, tienes garantía de que esas funciones no van a bloquear el programa. Puedes ver el programa completo (que se sigue pudiendo usar como cliente o servidor) en el Listado 15.10.

```

import sys
import socket
from select import select

SERVER = ('', 12345)
QUIT = b'bye'

class Chat:
    def __init__(self, sock, peer):
        self.sock = sock

```

```

        self.peer = peer

    def run(self):
        fds = [sys.stdin, self.sock]
        while 1:
            ready = select(fds, [], [])[0]
            for fd in ready:
                if self.sock in ready:
                    msg = self.receiving()
                else:
                    msg = self.sending()

            if msg == QUIT:
                return

    def sending(self):
        message = input().encode()
        self.sock.sendto(message, self.peer)
        return message

    def receiving(self):
        message, peer = self.sock.recvfrom(1024)
        print("other> {}".format(message.decode()))
        return message

if __name__ == '__main__':
    if len(sys.argv) != 2:
        print("usage: %s [--server|--client]" % sys.argv[0])
        sys.exit()

    mode = sys.argv[1]
    sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)

    if mode == '--server':
        sock.bind(SERVER)
        message, client = sock.recvfrom(0, socket.MSG_PEEK)
        Chat(sock, client).run()

    else:
        Chat(sock, SERVER).run()

```

LISTADO 15.10: Chat UDP (servidor y cliente) con select()
[📄/chat-udp/chat-select.py](#)

Aquí los métodos `sending()` y `receiving()` actúan como *manejadores* de los eventos de lectura de consola y del socket respectivamente. A diferencia de la versión con hilos, no hay bucles. Son funciones que se ejecutan cuando se detecta cada una de las condiciones, hacen una única lectura, procesan el mensaje y terminan.

Es sencillo adaptarlo para usar el módulo `selectors`, que indudablemente gana en limpieza, cambiando el método `run()` a algo como:

```

def run(self):
    selector = selectors.DefaultSelector()

```

```

selector.register(sys.stdin, selectors.EVENT_READ, self.sending)
selector.register(self.sock, selectors.EVENT_READ, self.receiving)

while 1:
    for key, mask in selector.select():
        if key.data() == QUIT:
            return

```

Puedes ver el código completo del servidor en el archivo `chat-udp/chat-selectors.py`.

15.3. Chatroom UDP

Ahora que están claros los mecanismos para el envío y recepción de mensajes, podemos plantear una solución general para un chat con múltiples usuarios. Llamaremos a esta modalidad ‘chatroom’ para distinguirlo de la versión para dos usuarios. Esta «sala de chat» es el *canal* al que los publicadores envían sus mensajes.

El nuevo servidor va a tomar el papel de *broker* que recibirá los mensajes de una cantidad arbitraria de participantes. Estos participantes serán a la vez publicadores, por su tarea de envío; y suscriptores, por su tarea de recepción.

La Figura 15.1 representa esta idea, pero ten en cuenta que los óvalos que representan a los participantes aparecen por duplicado, en sus dos roles de publicadores y suscriptores, para enfatizar el flujo de los mensajes.

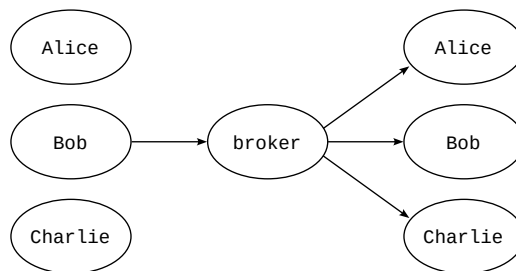


FIGURA 15.1: Arquitectura del chatroom

Puedes ver el código del broker en el Listado 15.11. Mantiene un diccionario llamado `members` que asocia el endpoint de cada participante con su *nick*. Al recibir el primer mensaje de un participante, lo añade al diccionario. Los siguientes mensajes ya se envían a todos los demás participantes precediendo el nombre del emisor. Por último, si el mensaje que envía el participante es `bye`, el broker lo elimina del diccionario y de la sala de chat.

Este programa es un buen ejemplo de cómo un socket UDP, puede operar múltiples flujos (es concurrente) a pesar de trabajar en un único hilo de ejecución.

Es interesante que el broker no necesita ningún mecanismo de concurrencia o gestión asíncrona porque solo usa un socket, el participante sí lo necesita (selectors en este caso) porque tiene dos entradas que atender: consola y socket.

```
import socket
QUIT = 'bye'

def main():
    sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
    sock.bind(('', 12345))
    members = {}

    while 1:
        data, endpoint = sock.recvfrom(1024)
        message = data.decode()
        if endpoint not in members.keys():
            members[endpoint] = message
            print("User '{}' has joined the chat.".format(message))
            continue

        sender = members[endpoint]
        for member_endpoint, nick in members.items():
            if member_endpoint == endpoint:
                continue

            print(nick, '<- ', message)
            encoded = "{}: {}\\n".format(sender, message).encode()
            sock.sendto(encoded, member_endpoint)

        if message == QUIT:
            del members[endpoint]
            print("User '{}' has left the chat.".format(sender))

    try:
        main()
    except KeyboardInterrupt:
        print("shut down.")
```

LISTADO 15.11: Broker de chatroom UDP
📄/chat-udp/chatroom-broker.py

El cliente (en el rol de participante) es en realidad muy parecido al del paso 4 del chat de dos usuarios (lo tienes en el archivo 📄/chat-udp/chatroom-member.py). Te animamos a probar el broker y unos cuantos participantes en terminales diferentes para que puedas comprobar cómo funciona.

Broker

```
$ ./chatroom-broker.py
New connection from ('127.0.0.1', 41656)
User 'alice' has joined the chat.
New connection from ('127.0.0.1', 53202)
User 'bob' has joined the chat.
New connection from ('127.0.0.1', 47864)
User 'charlie' has joined the chat.
alice <- hi everyone
bob <- hi everyone
```

Alice

```
$ ./chatroom-member.py 127.0.0.1 2000
alice
charlie: hi everyone
```

Bob

```
$ ./chatroom-member.py 127.0.0.1 2000
bob
charlie: hi everyone
```

Charlie

```
$ ./chatroom-member.py 127.0.0.1 2000
charlie
hi everyone
```

FIGURA 15.2: Chatroom con tres participantes

15.4. Chatroom TCP

Por supuesto también tiene mucho sentido hacer una versión TCP, no ya por rendimiento, si no por fiabilidad. Recuerda que UDP puede perder cualquier mensaje en cualquier momento, lo que no es lo que más conviene para una aplicación de mensajería.

Veamos las diferencias más importantes sobre el Listado 15.12.

```
import socket
import selectors

QUIT = 'bye'
PORT = 12345

class ChatroomBroker:
    def main(self):
        with socket.socket() as self.master:
            self.master.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
            self.master.bind(('', PORT))
            self.master.listen(5)

            self.members = {}
            self.selector = selectors.DefaultSelector()
            self.selector.register(self.master, selectors.EVENT_READ, self.acceptor)

            while True:
                for key, mask in self.selector.select():
                    key.data(key.fileobj)

    def acceptor(self, sock):
        conn, addr = self.master.accept()
        print("New connection from", addr)
        self.members[conn] = None
        self.selector.register(conn, selectors.EVENT_READ, self.receiver)
```

```

def receiver(self, conn):
    message = conn.recv(1024).decode().strip()
    if message == QUIT or not message:
        print("User '{}' has left the chat.".format(self.members[conn]))
        self.selector.unregister(conn)
        conn.close()
        del self.members[conn]
        return

    if not self.members[conn]:
        self.members[conn] = message
        print("User '{}' has joined the chat.".format(message))
        return

    sender = self.members[conn]
    for member_conn, nick in self.members.items():
        if member_conn == conn:
            continue

        print(nick, '<- ', message)
        encoded = "{}: {}\n".format(sender, message).encode()
        member_conn.sendall(encoded)

try:
    ChatroomBroker().main()
except KeyboardInterrupt:
    print("shut down.")

```

LISTADO 15.12: Broker de chatroom TCP
 chat-tcp/chatroom-broker.py

El broker mantiene el diccionario `members` cuya clave es el socket conectado y cuyo valor es el *nick* del participante. Lo hemos implementado con dos manejadores para un selector: `receiver()` y `reader()`. El primero, como su nombre indica, acepta cada nueva conexión, y la registra en el selector para recibir los mensajes de ese participante. El segundo es el que se encarga de esto. Tiene que manejar varias situaciones:

- Eliminación del participante cuando envía el mensaje `bye` o este cierra la conexión.
- Fijar el *nick* del participante cuando se recibe el primer mensaje.
- Reenviar a todos los demás participantes cuando se recibe un mensaje. En este caso, concatena el *nick* del emisor al comienzo del mensaje.

El participante también lo hemos implementado con un selector y dos manejadores: `sending()` y `receiving()`. El primero lee de la consola y envía el mensaje al broker, mientras que el segundo lee del socket y lo imprime en la consola. El código del participante se muestra en el Listado 15.13.

```

import sys
import socket
import selectors

```

```

QUIT = b'bye'

class ChatroomMember:
    def __init__(self, broker, nick):
        self.broker = broker
        self.nick = nick

    def sending(self):
        message = input().encode()
        self.sock.sendall(message)
        return message

    def receiving(self):
        message = self.sock.makefile().readline().strip()
        print(message)
        return message

    def run(self):
        self.sock = socket.socket()
        self.sock.connect(self.broker)
        self.sock.sendall(self.nick.encode())

        selector = selectors.DefaultSelector()
        selector.register(sys.stdin, selectors.EVENT_READ, self.sending)
        selector.register(self.sock, selectors.EVENT_READ, self.receiving)

        while 1:
            for key, mask in selector.select():
                if key.data() in [QUIT, '']:
                    return+33

if len(sys.argv) != 3:
    exit("Usage: ./chatroom-member.py <broker_address> <nick>")

try:
    broker = (sys.argv[1], 12345)
    ChatroomMember(broker, nick=sys.argv[2]).run()
except (KeyboardInterrupt, EOFError):
    print("shut down.")

```

LISTADO 15.13: Participante de chatroom
 chat-tcp/chatroom-member.py

Recuerda que con `selectors` o `select()` conseguimos un comportamiento asíncrono y concurrente (no paralelo) equivalente en ese sentido al broker UDP.

15.5. Chatroom con MQTT

En la sección anterior hemos visto como programar un broker TCP sencillo que se encarga de la distribución de mensajes desde los productores a los consumidores. MQTT es precisamente un protocolo diseñado para crear este tipo de aplicaciones. Las implementaciones de este protocolo suelen propor-

cionar un broker genérico. El más utilizado por mucho (y que también vamos a usar aquí) se llama *mosquitto*.

Para una sala de chat tan simple como la de la sección anterior, solo necesitamos escribir el código del participante (Listado 15.14) que hace uso de una librería cliente de MQTT llamada *paho-mqtt*, también con mucho la más usada en Python.

```
import json
import sys
import paho.mqtt.client as mqtt

TOPIC = 'chatroom'
QUIT = 'bye'

class ChatroomMember:
    def __init__(self, broker, nick):
        self.broker = broker
        self.nick = nick

    def on_connect(self, client, userdata, flags, reason_code, properties):
        self.client.subscribe(TOPIC)

    def on_message(self, client, userdata, msg):
        data = json.loads(msg.payload.decode())
        if data['nick'] == self.nick:
            return

        print("{}: {}".format(data['nick'], data['message']))

    def run(self):
        self.client = mqtt.Client(mqtt.CallbackAPIVersion.VERSION2)
        self.client.on_connect = self.on_connect
        self.client.on_message = self.on_message
        self.client.connect(*self.broker)
        self.client.loop_start()

        while 1:
            line = input()
            if line == QUIT:
                break

            payload = json.dumps({'nick': self.nick, 'message': line})
            self.client.publish(TOPIC, payload)

        self.client.loop_stop()
        self.client.disconnect()

if len(sys.argv) != 3:
    exit("Usage: ./chatroom-member.py <broker_address> <nick>")

try:
    broker = (sys.argv[1], 1883)
    ChatroomMember(broker, nick=sys.argv[2]).run()
except (KeyboardInterrupt, EOFError):
    print("shut down.")
```


LISTADO 15.14: Participante de chatroom con MQTT

 chat-mqtt/chatroom-member.py

Puedes apreciar lo mucho que se parece la estructura de este programa al del Listado 15.13. Hay un método `run()` en el que se configura un *callback* `on_connect()` que realiza la suscripción al canal chatroom y otro `on_message()` que se encarga de procesar e imprimir los mensajes recibidos.

Como puedes ver, con MQTT tanto los productores como los suscriptores utilizan la clase `Client`. En este caso, al igual que la versión TCP básica, el participante es a la vez publicador —invocando el método `publish()`— y suscriptor —con el callback de `on_message()`.

Este *callback* para `on_message()` funciona de un modo muy similar a los manejadores de *selectors*. Cuando hay datos, el *runtime* invoca el método y los pasa como argumento (*userdata*). Este mensaje puede ser de cualquier tipo que se pueda serializar como cadena o secuencia de bytes, pero lo más habitual con MQTT es usar JSON. A pesar de lo simple que es la carga útil de estos mensajes (nick, mensaje de usuario) hemos preferido usarlo también aquí.

El bucle de eventos de MQTT arranca con `loop_start()` y se ejecuta en un hilo independiente, por eso el método `run()` queda libre para la lectura de mensajes desde la entrada estándar.

El broker *mosquitto* se puede utilizar como un servicio del sistema, sin más que instalar el paquete oficial del mismo nombre. También es posible arrancarlo como un contenedor docker si lo prefieres con algo como:

```
$ docker run --rm -p 1883:1883 eclipse-mosquitto mosquitto -c /mosquitto-no-auth.conf
```

Para esta sencilla prueba, vamos a suponer que el broker está disponible como servicio del sistema en tu máquina. Puedes ejecutar varios participantes en varios terminales indicando la IP del broker y el nick de cada participante.

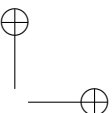
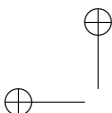
```
$ ./chatroom-member.py 127.0.0.1 alice
$ ./chatroom-member.py 127.0.0.1 bob
```

También hay un pequeño script *tmux*  chat-mqtt/run.sh que lanza 3 participantes e ilustra el funcionamiento del chatroom.

Y ¿qué más?

El modelo publicador-suscriptor es un muy común en aplicaciones distribuidas por su naturaleza asíncrona y desacoplada. Es especialmente útil

para aplicaciones de monitorización, notificación o actualización de estado como por ejemplo IoT. También es relevante su buen nivel de escalabilidad y adaptabilidad ya que permite que publicadores y suscriptores se añadan y eliminen sin afectar a los demás participantes. Por otro lado, la crítica más frecuente es que su broker representa un cuello de botella y un punto único de fallo. Solucionar esa limitación lleva arquitecturas más complejas basadas en broker federados o replicados, y en última instancia a arquitecturas sin broker, lo que da lugar a otro importante modelo de interacción: *peer-to-peer* (P2P), llamadas comunmente en español como *redes de pares*.



Capítulo 16

Calidad de servicio

Se dice de IP que es un protocolo *best effort*, que es una forma bonita para decir que hace lo que puede, pero podría ser nada en absoluto. La integridad, el orden o la propia entrega son prestaciones que IP consigue eventualmente, pero no están en absoluto aseguradas. Aunque por suerte TCP sí ofrece estas garantías, vamos a ver que hay otros aspectos que no puede resolver. Una de las limitaciones clave de la tecnología de conmutación de paquetes en la que se basa IP e Internet, es que no dispone de control de admisión, es decir, nunca niega el servicio a un nuevo usuario ni rechaza establecer una nuevo flujo a pesar de que la interred esté sufriendo ya una carga excesiva o incluso congestión.

El control de admisión es el aspecto más característico de las redes de conmutación de circuitos (como la telefonía). La red telefónica puede cuantificar la cantidad de recursos disponibles y rechazar el servicio solicitado por un cliente si estima que no lo va a poder satisfacer adecuadamente: el clásico «todas las líneas están ocupadas». El impacto del control de admisión es muy obvio y fácil de entender, pero veremos que hay otras formas de mejorar el servicio.

La Calidad de Servicio (QoS) engloba los mecanismos y tecnologías que intentan garantizar determinado nivel en los parámetros de tráfico para los servicios que ofrece una red de conmutación de paquetes. La idea que subyace a la QoS es que la red debería asignar sus recursos en función de las necesidades de cada usuario, flujo o servicio, bajo la premisa que no todos necesitan las mismas prestaciones. Por ejemplo, una llamada de voz requiere baja latencia, pero puede aceptar cierta pérdida de mensajes, mientras que la descarga de un archivo puede tolerar alta latencia, pero la pérdida de datos es inadmisibile.

Los parámetros básicos de la QoS son tasa de transferencia, latencia, *jitter* y tasa de pérdida de paquetes. Empezaremos este capítulo estudiando estos parámetros, su importancia, causas y consecuencias, la forma de medirlos y las técnicas que permiten controlarlos.

16.1. Tasa de transferencia

El primer parámetro que vamos a estudiar es la tasa de transferencia, que mide la velocidad con la que se mueven los datos ya sea a través de una red, de un enlace, un flujo UDP, una conexión TCP, etc. La tasa se mide en bits por segundo (b/s o bps) o bytes por segundo (B/s) y sus múltiplos (ver § D.3).

Se distingue habitualmente entre dos tipos de tasa:

Throughput

o tasa bruta, es la cantidad total de datos transmitidos en un período de tiempo. Además de la carga útil, incluye cabeceras, reconocimientos, retransmisiones o cualquier otro tipo de información de control; todo lo que llamamos sobrecarga de protocolos.

Goodput

o tasa neta, considera solo los datos efectivos, la carga útil procedente del usuario o la aplicación.

En realidad ambas tasas se pueden medir en distintas capas, considerando diferentes cabeceras y sus mecanismos de control. La tasa bruta de más bajo nivel debería contabilizar hasta las cabeceras de enlace, por lo que tiene que medirse directamente desde la interfaz de red. Veremos más adelante algunas herramientas, como el analizador de tráfico, que puede medirla. `FIXME(tshark, ss)`.

La tasa neta se suele medir sobre el nivel de transporte, es decir, con los datos que se envían o reciben con las primitivas del socket (como veremos en los ejemplos de código) o sobre el nivel de aplicación, que sería el caso en el que únicamente se cuentan los datos relevantes para el usuario, como por ejemplo, un fichero que se está descargando con HTTP.

También es útil distinguir la tasa en los extremos y en la red:

- **Tasa de envío** (T_s), es la velocidad con la que el proceso emisor entrega datos al SO a través de un socket, que como sabemos, en el caso de TCP no implica necesariamente que esos datos estén saliendo a la red y puede que simplemente se estén almacenando en el buffer de envío.
- **Tasa de recepción**¹ (T_r), es la velocidad con la que el proceso receptor recoge datos a través de un socket, que en realidad implica leer datos del buffer de recepción de la conexión.

¹en inglés: *drain rate*

- **Tasa de red²** (T_n), es la velocidad a la que los datos se mueven a través de la red. Esta tasa no se puede medir directamente, pero se puede estimar a partir de las tasas de envío y recepción.

Lógicamente para cualquiera de las tres pueden ser útiles tanto la tasa bruta como la neta. Los datos se envían en bloques (segmentos o datagramas) que pueden tener tamaños distintos y que son enviados/recibidos a intervalos posiblemente irregulares. Eso condiciona el cálculo y da lugar a distintos métodos. Veamos los más habituales.

16.1.1. Tasa instantánea

Es el cálculo más sencillo, y probablemente también el menos útil. Consiste en contabilizar únicamente los datos transmitidos desde la medición anterior considerando el tiempo transcurrido desde entonces. Formalmente:

$$R_{\text{inst}} = \frac{\Delta B}{\Delta t} \quad (16.1)$$

donde: R_{inst} es la tasa instantánea (bytes por segundo), ΔB es la cantidad de datos transmitida en el último bloque (bytes), y Δt es el tiempo transcurrido desde el envío del bloque anterior (segundos).

La tasa instantánea puede variar drásticamente de un mensaje al siguiente. Por eso se dice que es muy «sensible al ruido», y por ello raramente resulta útil.

Por ejemplo, en Python la tasa de envío neta instantánea podría calcularse como aparece en el siguiente fragmento de código³. Fíjate que usamos `time.monotonic()` en lugar de `time.time()` para que el cálculo de los delta no se vea afectado por ajustes en el reloj del sistema.

```
while 1:
    last_time = time.monotonic()
    data = get_data()
    sock.sendall(data)
    byte_rate = len(sent) / time.monotonic() - last_time
    print(byte_rate)
```

16.1.2. Promedio acumulado (CA)

En el otro extremo de la tasa instantánea está el promedio total acumulado o CA (Cumulative Average). Consiste simplemente en calcular la media de toda la transmisión hasta el momento, es decir, el total de datos transmitidos partido por el tiempo transcurrido desde el inicio.

²en inglés: *network rate*

³Adaptarlo a la tasa de recepción es directo

$$R_{\text{avg}}(t) = \frac{B(t) - B(t_0)}{t - t_0} \quad (16.2)$$

donde: $R_{\text{avg}}(t)$ es la tasa media acumulada (bytes por segundo), $B(t)$ es la cantidad total de datos transmitida hasta ahora (bytes), $B(t_0)$ es la cantidad total de datos transmitida en el instante 0, y $t - t_0$ es el tiempo transcurrido desde el comienzo (segundos).

Lógicamente sufre del efecto contrario a la tasa instantánea. No se adapta en absoluto a las variaciones (nada reactivo), por lo que una reducción o aumento momentáneo de la tasa queda completamente enmascarado.

En Python:

```
start_time = time.monotonic()
sent = 0
while 1:
    data = get_data()
    sock.sendall(data)
    sent += len(data)
    elapsed = time.monotonic() - start_time
    byte_rate = sent / elapsed
    print(byte_rate)
```

LISTADO 16.1: Promedio acumulado

16.1.3. Media móvil simple (SMA)

Una técnica intermedia que ofrece información actualizada, pero más estable que la tasa instantánea, es la ‘media móvil simple’ o SMA (Simple Moving Average). Se basa en definir una ventana de tiempo que abarca solo los w segundos previos considerando únicamente los datos transmitidos en ese lapso.

$$R_{\text{SMA}}(t) = \frac{B(t) - B(t - w)}{w} \quad (16.3)$$

donde: $R_{\text{SMA}}(t)$ es la tasa media móvil simple (bytes por segundo), $B(t)$ es la cantidad total de datos transmitida hasta ahora (bytes), $B(t - W)$ es la cantidad total de datos transmitida hace w segundos, y w es el tamaño de la ventana (segundos).

Aunque es una medida mucho más reactiva que el promedio acumulado, los nuevos datos tardan algún tiempo ($w/2$ segundos) en reflejarse en la tasa.

En Python:

```

class SMA_RateMeter:
    def __init__(self, window_size=5):
        self.window_size = window_size
        self.window = collections.deque()

    def update(self, sent_bytes):
        now = time.monotonic()
        self.window.append((now, sent_bytes))

        # Eliminar datos fuera de la ventana
        while self.window and (now - self.window[0][0]) > self.window_size:
            self.window.popleft()

    @property
    def rate(self):
        if not self.window:
            return 0

        window_bytes = sum(b for t, b in self.window)
        elapsed = self.window[-1][0] - self.window[0][0]
        return window_bytes / elapsed if elapsed > 0 else 0

sma = SMA_RateMeter(window_size=5)
while 1:
    data = get_data()
    sock.sendall(data)
    sma.update(len(data))
    print(sma.rate)

```

16.1.4. Media móvil exponencial (EMA)

SMA también tiene un problema: los datos nuevos de la ventana tienen el mismo peso que los antiguos, por lo que la tasa no refleja bien las variaciones en el tráfico (poco reactivo). Para paliar ese efecto se puede usar una ‘media móvil exponencial’, o EMA (Exponential Moving Average), que aplica un decaimiento exponencial, asignando menos peso a los datos más antiguos.

Se calcula utilizando la siguiente fórmula:

$$R_{\text{EMA}}(t) = \alpha \cdot R_{\text{inst}}(t) + (1 - \alpha) \cdot R_{\text{EMA}}(t - 1) \quad (16.4)$$

donde: $R_{\text{EMA}}(t)$ es la tasa media móvil exponencial, $R_{\text{inst}}(t)$ es la tasa instantánea en el instante t , y α es el factor de suavizado ($0 < \alpha < 1$).

El factor de suavizado α permite ajustar la reactividad del cálculo. Así un valor de 1 la hace equivalente a la tasa instantánea, y valores menores aumentan el peso de los cálculos anteriores. Un valor típico es $\alpha = 0,1$.

En Python, una implementación básica del EMA podría ser:

```

class EMA_RateMeter:
    def __init__(self, alpha=0.1):
        self.alpha = alpha
        self.rate = 0
        self.last = time.monotonic()

    def update(self, sent_bytes):
        elapsed = time.monotonic() - self.last
        self.rate = self.alpha * sent_bytes / elapsed + (1 - self.alpha) * self.rate
        self.last = time.monotonic()

ema = EMA_RateMeter()
while 1:
    data = get_data()
    sock.sendall(data)
    ema.update(len(data))
    print(ema.rate)

```

16.1.5. Consideraciones sobre el cálculo de tasa

Tampoco EMA es perfecto. Al comienzo de una conexión TCP, el emisor puede enviar —invocando `send()`— una gran cantidad de datos en muy poco tiempo porque en realidad esos datos se almacenan en el buffer de envío y recepción, pero no están siendo aún consumidos realmente por el proceso receptor.

Esto genera cálculos de tasa instantánea (en los que se basa EMA) enormes e irreales, porque a fin de cuentas lo que se está midiendo en esos primeros instantes es la tasa entre el proceso y la memoria local y remota, pero no entre los procesos emisor y receptor, que es lo que proporciona una medida representativa. Estos valores tan altos falsean el resultado, e incluso con el decaimiento exponencial, puede llevar mucho tiempo compensar ese efecto. En esta situación no mejora la reactividad, que acaba siendo tan pobre como la del promedio acumulado.

Hay algunas opciones que pueden ayudar a paliar este problema:

- Ignorar la tasa instantánea los primeros segundos de la conexión (período de calentamiento o *warmup*).
- Descartar lapsos (Δt) demasiado pequeños.
- Utilizar un α bajo para Δt pequeños. Se puede calcular como $\alpha = 1 - e^{-dt/\tau}$ siendo τ una constante que determina la reactividad.
- Reducir el tamaño de los buffers TCP de envío y recepción.
- No usar EMA en el emisor.
- Usar SMA en lugar de EMA durante el calentamiento o si no se necesita alta reactividad.

Aplicaremos algunas de estas técnicas a los ejemplos de este capítulo.

16.2. Control de flujo TCP como limitador de tasa

Sabemos que el control de flujo TCP permite al receptor controlar la cantidad de datos que el emisor le envía de acuerdo al espacio del que dispone (la ventana que anuncia). Aunque no es su objetivo principal, este mecanismo se puede utilizar para conseguir un control de la tasa de transferencia rudimentario. Veamos cómo.

Considerando solo uno de los sentidos de la comunicación, asumamos por un momento que el emisor siempre tiene datos disponibles y los puede enviar siempre que sea posible. Con esta premisa se pueden dar tres situaciones en función de la tasa de recepción (T_r) respecto a la velocidad de la red (T_n). Lógicamente la tasa de emisión (T_s) estará limitada por la menor de las otras dos.

- Si $T_r < T_n$, el buffer de recepción se llena, la ventana se cierra, el buffer de emisión también se llena y el emisor se bloquea (*stall*) en espera de que se libere espacio cuando la ventana vuelva a abrir. En esta situación, la tasa de emisión está limitada por el proceso receptor.
- Si $T_r > T_n$, el buffer de recepción queda vacío y el receptor quedará bloqueado en espera de nuevos datos. En esta situación, la tasa de emisión está limitada por la red.
- Si $T_r \approx T_n$, el buffer de recepción nunca se llena ni queda vacío. Ni el emisor ni el receptor se bloquean.

Lógicamente, los buffers de emisión y recepción existen precisamente porque el tercer caso, aunque deseable, es poco frecuente. Estas tres velocidades: emisión, red y recepción suelen ser dispares y cambiantes. Los buffers amortiguan esas diferencias y permiten un flujo más estable. Sin embargo, obviando otros controles —como el de congestión— cuando la ventana está abierta el emisor TCP envía datos a la máxima tasa posible en ese momento, y eso frecuentemente provoca bloqueos en ambos extremos, lo que influye en parte en que T_r y T_n varíen constantemente. Además el algoritmo de Nagle trata de evitar segmentos pequeños, lo que alarga esos bloqueos. La conclusión es que cuando existe disparidad en el desempeño de emisor y receptor, el mecanismo de control de flujo (y en menor medida el de congestión) provoca un flujo a ráfagas, es decir, momentos en los que se envían datos a la máxima velocidad posible y momentos en los que no se envía nada (porque la ventana está cerrada).

La consecuencia directa de todo esto es que si el receptor limita intencionadamente la tasa de recepción, la tasa media de emisión se verá también limitada, aunque su dispersión (*p. ej.* la desviación típica) será muy alta debido al flujo en ráfagas.

En principio una limitación de tasa de este tipo no es incorrecta, pero cuando ocurre de forma generalizada puede provocar efectos perjudiciales como el aumento de la latencia o el *jitter*. Lo veremos más adelante.

16.3. Limitación de tasa en el receptor

Veamos cómo implementar un control básico de tasa en el receptor. Como acabamos de ver, se puede conseguir bloqueando intencionadamente el proceso con `time.sleep()` el tiempo necesario para aproximar la tasa medida a la tasa deseada. El siguiente código calcula la tasa con el método de promedio acumulado (`ca_rate`), pero puedes cambiarlo por alguna de las otras opciones que acabamos de ver si lo prefieres.

```
1 target_rate_kBps = 200 * 1000 # 200 kB/s
2 start_time = time.monotonic()
3 received = 0
4 while 1:
5     data = sock.recv(4096)
6     if not data:
7         break
8
9     received += len(data)
10    elapsed = time.monotonic() - start_time
11    ca_rate = received / elapsed
12
13    if ca_rate <= target_rate_kBps:
14        continue
15
16    adjust_time = (received / target_rate_kBps) - elapsed
17    if adjust_time > 0:
18        time.sleep(adjust_time)
```

LISTADO 16.2: Limitación de la tasa de recepción
(medida con promedio acumulado)

Hasta la **línea 14** es equivalente al Listado 16.1 salvo que aquí estamos calculando la tasa de recepción. Si este cálculo está por debajo de la tasa objetivo (`target_rate_kBps`) no hace nada, pero si está por encima, realiza un ajuste aumentando el tiempo (el denominador de la fórmula 16.2). El ajuste se obtiene como la diferencia entre el tiempo realmente transcurrido y el tiempo que corresponde a la tasa objetivo (**línea 16**).

Vamos a retomar el ejemplo del directorio `flow-control` que ya utilizamos en el capítulo 11 para ilustrar cómo lograrlo. El emisor (cliente) envía datos al receptor (servidor) tan rápido como puede, pero el receptor limita la tasa aplicando la idea que acabamos de ver. Ambos programas calculan la tasa de envío y recepción respectivamente mediante promedio acumulado (§ 16.1.2).

Puedes probar el ejemplo con los siguientes comandos. El segundo parámetro del servidor es la tasa máxima deseada: 200 kB/s.

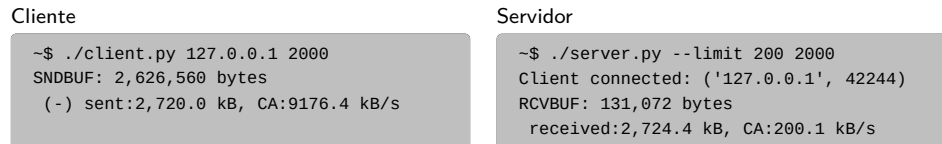


FIGURA 16.1: Limitación de la tasa en el receptor
 flow-control

Si lo ejecutas podrás ver que la transferencia en el lado del cliente se detiene durante unos segundos de vez en cuando tal como se ha explicado en la sección anterior. El receptor no para en ningún momento porque sigue consumiendo los datos que hay en el buffer de recepción. Cuando se libera suficiente espacio, la ventana se abre y el emisor envía nuevos datos durante un tiempo, hasta que el buffer se llena y la ventana se cierra de nuevo.

La Figura 16.2 es un esquema de su funcionamiento y la Figura 16.3 muestra la cantidad de datos enviados por el emisor a partir de la información obtenida de su ejecución (almacenada en el archivo `client-stats.csv`). El diagrama en escalera permite apreciar claramente los períodos de bloqueo (*stall*) debidos al cierre de la ventana.

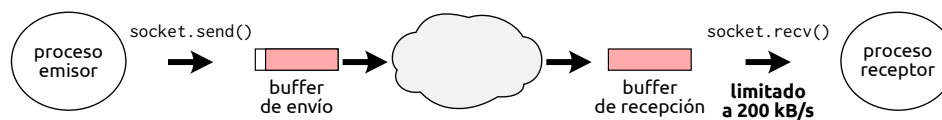


FIGURA 16.2: Limitación de tasa en el receptor

Aquí puedes apreciar perfectamente el efecto del calentamiento (*warmup*) del que hablábamos en §16.1.5 respecto a la evolución de la tasa. Debido a que los buffers de envío y recepción son grandes: 2624 kB y 131 kB respectivamente⁴, el emisor al comienzo puede enviar muchos datos (~2,7 MB) muy rápido. Una vez llenos los buffers, bloquea por el cierre de la ventana. Con el tiempo la tasa media se irá aproximando a los 200 kB/s impuestos por el receptor. Puedes ver esa evolución en la gráfica 16.4 obtenida también a partir de los datos generados por el emisor.

Los dientes de sierra se deben a que el emisor envía muchos datos en poco tiempo y bloquea (rampa ascendente); cuando reanuda, han pasado unos

⁴Esos son los valores por defecto en GNU/Linux para `localhost`.

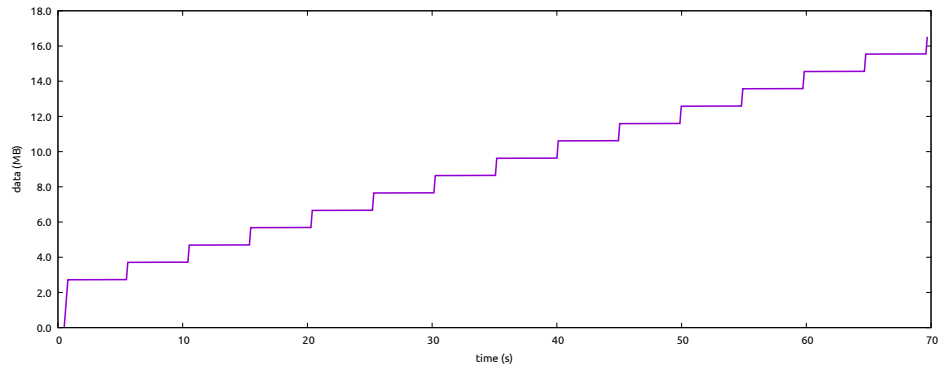


FIGURA 16.3: Datos enviados por el emisor

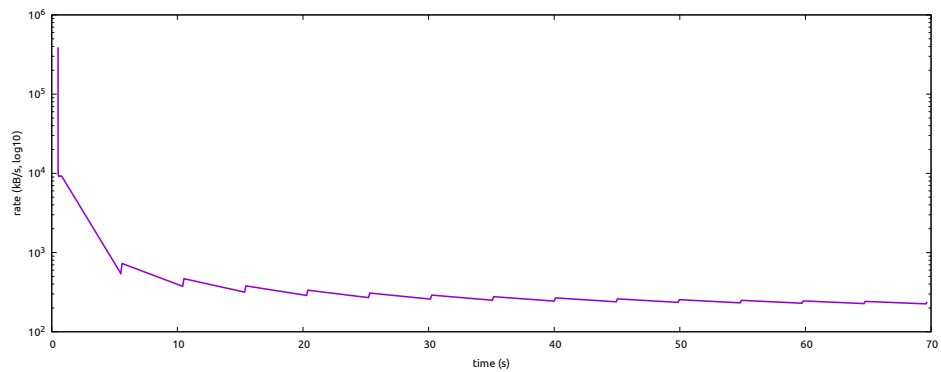


FIGURA 16.4: Tasa de envío medida en el emisor (promedio acumulado)

segundos por lo que la tasa media baja un poco (rampa descendente). Usando otro método para el cálculo de tasa, lógicamente obtendríamos gráficas sin este efecto.

Este mismo servidor proporciona alternativamente un modo diferente de trabajo (que se activa con `-step`). Este argumento permite especificar una cantidad de bytes, que una vez recibidos hacen que el servidor se detenga en espera de que el usuario pulse **ENTER** para continuar. De ese modo puedes comprobar fácilmente cómo el emisor bloquea al llenarse los buffers de envío y recepción y no continúa hasta que quede espacio libre en el receptor. Esta lógica está implementada en el método `step_receiving()` (Listado 16.3).

```

159     def step_receiving(self):
160         input('Press ENTER to receive > ')
161         received = 0
162         while 1:
163             count = 0

```

```

164         pending = self.step_size * 1000
165         while count < pending:
166             data = self.conn.recv(min(4096, pending - count))
167             if not data:
168                 return
169             count += len(data)
170             received += count
171
172         input(f'received: {received//1000:,} kB > ')

```

LISTADO 16.3: Servidor con control manual de la recepción
 flow-control/server.py

En los ejemplos previos el cliente simplemente envía datos sin sentido, y el servidor los descarta. Sin embargo, tienen otro modo que permite indicar al cliente que consuma datos desde su entrada estándar (con `--stdin`) y al servidor que envíe los datos recibidos a su salida estándar (con `--stdout`). De este modo podemos crear un sistema sencillo para transmitir un fichero mp3 a través de la red, que será consumido en el servidor directamente por un reproductor de audio, vía redirección. Obviamente para que funcione correctamente, el cliente debe enviar un archivo adecuado (un mp3). Este escenario es interesante porque es el reproductor el que determina la tasa de recepción en función de la calidad y compresión del archivo (su *bitrate*). Puedes probar esa posibilidad con estos comandos:

```

~$ ./server.py --stdout --rcvbuf 4000 2000 | mpg123 -q -
~$ ./client.py --stdin --sndbuf 4000 < audio.mp3

```

FIGURA 16.5: Tasa limitada por el reproductor de mp3
 flow-control-player

Fíjate en las opciones `--sndbuf` y `--rcvbuf` que establecen el tamaño de los respectivos buffer de envío y recepción. Estos valores favorecen períodos de bloqueo más cortos que ayudan a entender cómo funciona la transferencia.

Tanto el cliente como el servidor muestran la tasa con promedio acumulado (CA). El servidor además puede mostrar la tasa SMA si se activa el argumento `--sma` y la tasa EMA con compensación de calentamiento basada en el ajuste de α respecto Δt si se activa el argumento `--ema`. La Figura 16.7 representa ambas tasas para comparación.

En la ejecución de la Figura 16.5 la tasa de recepción ronda los 41 kB/s (328 kbps), muy próxima al bitrate con el que está codificado el mp3 con el que hemos realizado la prueba: 320 kbps.

```
flow-control$$ ./server.py --ema --sma --stdout --rcvbuf 4000 2000 | mpg123 -q -
Client connected: ('127.0.0.1', 52406)
RCVBUF: 8,000 bytes
received:968.7 kB, CA:43.0 kB/s, EMA:40.8 kB/s, SMA:41.0 kB/s
```

FIGURA 16.6: Opciones para medición de tasa
en el servidor: EMA y SMA
🔗/flow-control-player/server.py

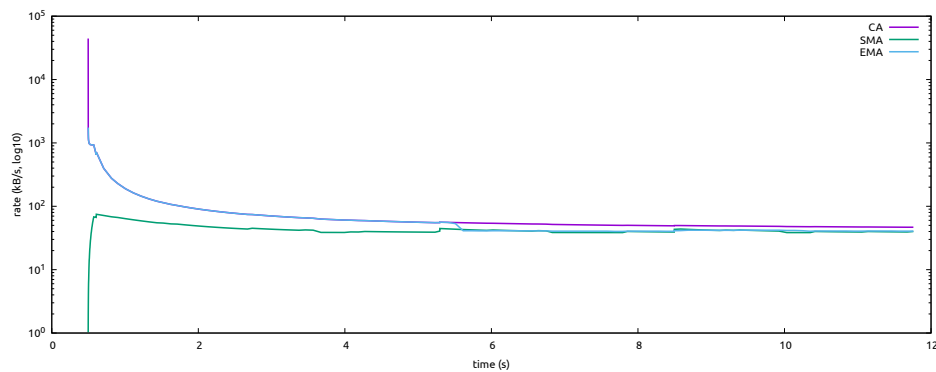


FIGURA 16.7: Comparación de CA, SMA y EMA en el servidor

La consecuencia de tener el reproductor en el receptor es la misma que en los otros ejemplos. Como la tasa de recepción es menor que la tasa de emisión, ambos buffers se llenan y el emisor queda eventualmente bloqueado de manera intermitente. En este caso, también el receptor puede quedar bloqueado porque la entrada estándar del reproductor dispone de un buffer adicional, con lo que la llamada `stdout.flush()` puede bloquear el proceso receptor hasta que haya espacio libre en ese buffer. Ten en cuenta que esto ocurre porque ambos cliente y servidor se están ejecutando en el mismo nodo y se comunican a través de `localhost`. En una conexión remota y en función del ancho de banda disponible, este comportamiento puede variar mucho, hasta el punto de no ocurrir bloqueos si ese ancho de banda es similar al bitrate al que el reproductor consume el flujo.

```
~$ file audio.mp3
audio.mp3: Audio file with ID3 version 2.4.0, contains: MPEG ADTS, layer III, v1,
320 kbps, 44.1 kHz, JntStereo
```

FIGURA 16.8: Información del mp3 utilizado en la prueba

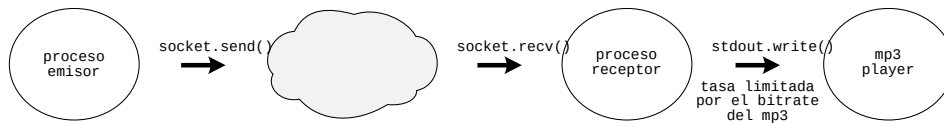


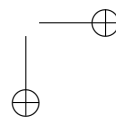
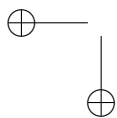
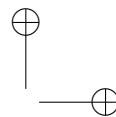
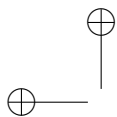
FIGURA 16.9: Datos enviados por el emisor

16.4. Limitación de tasa en el emisor

Por supuesto, el emisor puede aplicar exactamente la misma técnica intercalando pausas para conseguir una tasa de emisión concreta. Sin embargo, en este caso la tasa será mucho más estable. El emisor puede enviar bloques de datos más pequeños más a menudo, lo que reduce mucho la dispersión.

El problema es que el emisor no sabe a qué tasa consume el receptor. Si se excede, en algún momento se llenarán los buffers y se bloqueará el emisor. Si es demasiado baja, será el receptor el que se bloquee en espera de datos nuevos y la reproducción del audio se interrumpirá.

[Capítulo incompleto]



Capítulo 17

Serialización

Al terminar este capítulo, entenderás:

- Qué es la serialización y la des-serialización.
- Por qué es importante definir un formato de datos común.
- Cómo es la serialización binaria de datos multibyte y texto.
- Cómo utilizar struct para manejar datos binarios simples de tipos diferentes, como por ejemplo una cabecera de mensaje binario.

Empezamos este capítulo con una afirmación que podría aparecer en un curso de informática para niños: «La única forma en la que un computador digital procesa y almacena datos es mediante código binario». Ya, lo tenemos claro, pero eso también tiene consecuencias cuando esos datos tienen que viajar por la red.

El problema es que, salvo en unas pocas excepciones, el binario rara vez resulta suficientemente expresivo o amigable 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 necesita: enteros, decimales, cadenas de caracteres, fechas, etc. Lo importante es recordar que, sea cual sea su representación en un lenguaje de programación de alto nivel, en la memoria o registros del computador, todo dato es a fin de cuentas una secuencia de bits.

La **serialización** es el proceso que transforma los datos que manejan los programas (enteros, cadenas, imágenes, etc.) en secuencias de bytes susceptibles de ser almacenadas en un archivo o enviadas a través de la red. La des-serialización es el proceso inverso. Existen dos tipos de serialización: binaria y textual. La binaria codifica los datos como secuencias de bytes, mientras que la textual utiliza marcado legible por humanos¹, con identificadores, espacios y comillas.

¹El significado del adjetivo «legible» es bastante debatible según qué protocolo.

El siguiente listado es un ejemplo de serialización binaria (con pickle) y textual (con JSON) de los mismos datos:

Binaria

```
>>> import pickle
>>> data = {'width': 10, 'height': 20}
>>> pickle.dumps(data)
b'\x80\x04\x95\x1a\x00\x00\x00\x00...
```

Textual

```
>>> import json
>>> data = {'width': 10, 'height': 20}
>>> json.dumps(data)
'{"width": 10, "height": 20}'
```

FIGURA 17.1: Serialización binaria y textual

Por supuesto existen muchos formatos o sistemas de codificación tanto binaria como textual. En este capítulo veremos algunos de los más sencillos para ilustrar el concepto con mínima complejidad.

Es buen momento para aclarar una confusión habitual con la terminología. No debes 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 ninguna clave, ocultación u ofuscación del mensaje para conseguir confidencialidad. Por ejemplo, un mensaje codificado en *código* Morse es perfectamente entendible por cualquier persona o sistema que conozca el código, que por supuesto, es público. El cifrado pretende ocultar el contenido del mensaje, para que únicamente aquellos que conozcan la clave secreta puedan verlo. Un mensaje cifrado con Blowfish solo podrá ser descifrado por alguien que conozca la clave.

17.1. Representación, sólo eso

Una de las excepciones en las que el binario es útil directamente es la *programación de sistemas*, es decir, aquellos programas que consumen directamente servicios del SO. El binario resulta útil para manejar campos de bits, flags (banderas binarias) o máscaras, muy comunes cuando se manipulan registros de control, operaciones de E/S, etc. Por eso cualquier programador también debería manejar con soltura la representación binaria.

La codificación más básica y común es la que aplica una base numérica. Los lenguajes de programación, como las personas, utilizan la base decimal para expresar cantidades. Python permite expresar literales numéricos en varias bases. Todos los casos del siguiente ejemplo representan 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. Puedes comprobarlo fácilmente en el Listado 17.1, que se está ejecutando en el modo interactivo del intérprete.

```
>>> 0b101010 # binario
42
>>> 0o52      # octal
42
>>> 0x2A      # hexadecimal
42
```

LISTADO 17.1: Literales numéricos en Python

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`) porque su objetivo es precisamente ofrecer diferentes *representaciones textuales* del mismo dato. Observa las comillas simples en los valores de retorno en el Listado 17.2 que indican claramente que se trata de cadenas.

```
>>> bin(42)
'0b101010'
>>> oct(42)
'0o52'
>>> hex(42)
'0x2A'
```

LISTADO 17.2: Conversión a representación binaria, octal y hexadecimal

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). Puedes verlo en el Listado 17.3.

```
>>> int('42')
42
>>> int('52', 8)
42
>>> int('1J', 23)
42
```

LISTADO 17.3: Especificando la base en el constructor de `int`

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

17.2. 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]`. Sin embargo, Python no dispone de ese tipo². Python, por su naturaleza dinámica, solo

²No lo debes confundir con `bytes`, que es un tipo de secuencia.

tiene un tipo de datos para enteros: `int`. Estos enteros pueden ser arbitrariamente grandes puesto que el *runtime* se encarga de gestionar la memoria necesaria:

```
>>> googol = 10 ** 100
>>> type(googol)
<class 'int'>
```

Pero cuando se serializan datos en un archivo o una conexión de red, necesitamos precisar explícitamente el tamaño de los datos. Aunque Python no disponga de los tipos `byte`, `short`, `long`, etc., usaremos estos conceptos para entender los tamaños de los datos. Veremos cómo se logra todo esto en las siguientes secciones.

17.3. Caracteres

La codificación de caracteres más simple consiste en asignar un número a cada carácter del alfabeto. El código más común y veterano es ASCII, que fue creado por ANSI en 1963 como una evolución de la codificación utilizada anteriormente en telegrafía. Es un código de 7 bits (128 símbolos) que incluye los caracteres 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 se crearon extensiones, como por ejemplo ISO 8859-1, popularmente conocida como *latin-1* que dispone de 256 símbolos, entre los que se encuentran caracteres acentuados y otros símbolos de uso común en idiomas europeos (ß, ç, ñ, ¿, etc.), monedas (£, ¥) o signos matemáticos ($\pm$, $\div$, $\times$, $\frac{1}{4}$, etc.).



El carácter «retorno de carro» (CR, código 10 o 0x0A), pide mover el cursor a la primera columna (en el borde izquierdo), mientras que el carácter de «avance de línea» (LF, código 13 o 0x0D) pide mover 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 físico para comenzar a escribir la siguiente línea del papel. Las computadoras, por analogía, utilizaban la secuencia CR-LF para indicar la misma operación en un terminal (una pantalla). Sigue siendo de este modo en los sistemas operativos de Microsoft a día de hoy. Por contra, los creadores de los sistemas UNIX entendieron que el salto de línea sin retorno de carro no tenía sentido en un terminal 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.

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

```

LISTADO 17.4: Conversión entre caracteres y enteros en Python

Fíjate en la **línea 5** que el código del **carácter** `\0` es 48, mientras que (**línea 7**) el carácter equivalente al código 0 es `\x00`. Es especialmente importante tener claro que los caracteres numéricos **no corresponden** con 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.

```

>>> for i in '\xd2a3\n':
...     print ord(i)
210
97
51
10

```

La cadena `\xd2a3\n` está compuesta por los caracteres `\xd2`, `a`, `3` y `\n`.

No puedes manipular el contenido de una cadena; recuerda que el tipo `str` es inmutable. Sin embargo, tenemos el tipo `bytearray`, que puede almacenar una secuencia de bytes, modificar su contenido —acepta tanto caracteres

como enteros— y permite obtener fácilmente la lista de caracteres o secuencia de enteros equivalente:

```
>>> buf = bytearray('abcd', 'ascii')
>>> buf[0] = 20
>>> buf
bytearray(b'\x14bcd')
>>> buf.decode()
'\x14bcd'
>>> bytes(buf)
b'\x14bcd'
>>> list(buf)
[20, 98, 99, 100]
```

17.4. Tipos multibyte y ordenamiento

Como sabes, un único 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 «caben» en un byte. El más sencillo de estos tipos es el *short* o entero de 16 bits³. 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 forman un *short*?

Dependiendo de la respuesta se distingue entre *little endian* y *big endian*. *Little endian* significa que el byte de *menor* peso⁴ se coloca en la dirección más baja de memoria, mientras que en *big endian* es el byte de *mayor* peso⁵. La Figura 17.2 lo muestra para un dato de 4 bytes.

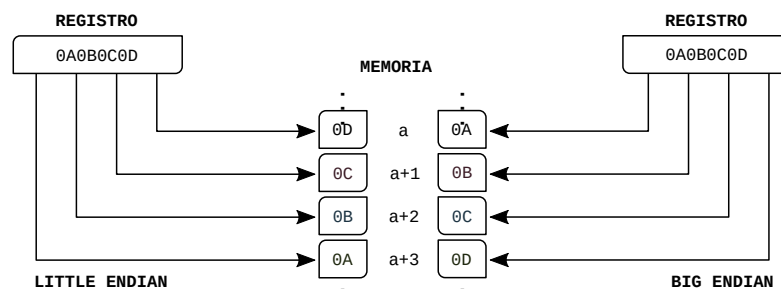


FIGURA 17.2: Ordenación de bytes

³short tampoco existe como tipo nativo en Python.

⁴el que tiene los bits menos significativos (LSB)

⁵el MSB

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.

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

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

LISTADO 17.5: Averiguar el ordenamiento de bytes con Python.

Aunque Python ofrece una forma directa:

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

Algo parecido ocurre también con la red. Cuando se coloca un dato multi-byte *en el cable*⁶ debe elegirse un ordenamiento. Pues bien, los protocolos de la pila TCP/IP utilizan siempre ordenamiento *big-endian* [33], es decir, se envía primero el byte más significativo. Y de hecho, se le conoce como *ordenamiento de la red* (*network byte order*).

Eso significa que las arquitecturas *little-endian* (típicamente Intel y AMD) deben convertir sus datos multi-byte antes de enviarlos a la red sobre un socket. Para evita que el programa tenga que comprobar por sí mismo qué ordenamiento utiliza la arquitectura en la que se está ejecutando, las librerías de sockets proporcionan funciones que realizan la conversión. Nótese que en un nodo *big-endian* estas funciones no harán nada⁷, pero deben usarse para conseguir que el programa sea ‘portable’, es decir, que se pueda ejecutar en nodos de diferente arquitectura sin ninguna modificación.

Las funciones que ofrece Python, 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: ***network to host short***.
- `socket.ntohl()`. Convierte un entero de 32 bits (*long*) del ordenamiento de la red al del host: ***network to host long***.
- `socket.htons()`. Convierte un entero de 16 bits (*short*) del ordenamiento del host al de la red: ***host to network short***.

⁶por analogía con la expresión en inglés *on the wire*

⁷retornan el mismo valor que se les pasa como parámetro

- `socket.htonl()`. Convierte un entero de 32 bits (*long*) del ordenamiento de host al de la red: *host to network long*.

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

```
>>> socket.htons(32)
8192
>>> socket.htonl(32)
536870912
>>> socket.htons(0)
0
```

LISTADO 17.6: Funciones de conversión de ordenamiento del módulo `socket`.

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

```
>>> hex(32)
'0x20'
>>> hex(socket.htons(32))
'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 cambios.

17.5. Cadenas de caracteres y secuencias de bytes

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

```
>>> string = "hello world"
>>> type(string)
<class 'str'>
>>> sequence = bytes(string, 'ascii')
>>> type(sequence)
<class 'bytes'>
>>> string
'hello world'
>>> sequence
b'hello world'
```

LISTADO 17.7: Cadena codificada en ASCII



Python puede utilizar muchos sistemas de codificación (*encodings*) diferentes. El más habitual en los sistemas POSIX es UTF8 y también lo es para Python. UTF8 puede representar cualquier carácter Unicode. Es compatible con ASCII para su rango (127 caracteres), es decir, requiere un byte, mientras que para el resto de caracteres necesita 2 o más bytes. Por ejemplo el carácter ñ se codifica con 2 bytes: '0xc3b1' y € se codifica con 3 bytes: '0xe282ac'.

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 este ejemplo usa codificación «ascii». ASCII codifica cada carácter con un único byte, por eso coinciden. Veamos otro ejemplo más ilustrativo:

```
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
```

LISTADO 17.8: Cadena codificada en UTF8

Esta vez, al intentar codificar la cadena «ñandú» con el *encoding* ASCII se produce un error (**línea 2**), porque ASCII no puede representar las letras ñ y ú. Pero UTF8 sí que puede, aunque requiere 2 bytes para esos caracteres 'no ASCII'. 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:

```
>>> bytes('ñandú', 'utf-8')
b'\xc3\xba\xc3\xba'
>>> 'ñandú'.encode('utf-8')
b'\xc3\xba\xc3\xba'
>>> str(b'\xc3\xba\xc3\xba', 'utf-8')
'ñandú'
>>> b'\xc3\xba\xc3\xba'.decode('utf-8')
'ñandú'
```

LISTADO 17.9: Constructores y métodos `str.encode()` y `bytes.decode()`

17.6. Empaquetado

El módulo `struct` de la librería estándar de Python puede hacer transformaciones hacia y desde binario de un modo mucho más flexible y, sobre todo, cómodo. La función `struct.pack()`⁸ *serializa* datos nativos de Python generando una secuencia de bytes (tipo `bytes`) según la especificación de tamaño y ordenamiento que se le indique. Por ejemplo, el siguiente listado serializa el número 5 como un entero de 32 bits con ordenamiento *big-endian* y después como *little-endian*:

```
>>> struct.pack('>i', 5)
b'\x00\x00\x00\x05'
>>> struct.pack('<i', 5)
b'\x05\x00\x00\x00'
```

LISTADO 17.10: `struct`: alternativas de ordenamiento de un entero de 32 bits

El primer parámetro de `pack()` es la especificación de la conversión. Hay dos conjuntos de símbolos: uno para especificar ordenamiento (ver tabla La tabla 17.1) y otro para especificar formato (ver tabla 17.2).

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.

```
>>> struct.pack('>h', 5)
b'\x00\x05'
>>> struct.pack('<h', 5)
b'\x05\x00'
```

LISTADO 17.11: `struct`: alternativas de ordenamiento de un entero de 16 bits

@	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 17.1: `struct`: especificación de ordenamiento

El Listado 17.12 muestra el resultado de aplicar los diferentes formatos al mismo dato en un computador *little-endian*.

```
>>> struct.pack('b', 5)
b'\x05'
>>> struct.pack('?', 5)
```

⁸empaquetar

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 17.2: struct: especificación de formato

```

b'\x01'
>>> struct.pack('h', 5)
b'\x05\x00'
>>> struct.pack('i', 5)
b'\x05\x00\x00\x00'
>>> struct.pack('l', 5)
b'\x05\x00\x00\x00\x00\x00\x00\x00'
>>> struct.pack('f', 5)
b'\x00\x00\xa0@'
>>> struct.pack('d', 5)
b'\x00\x00\x00\x00\x00\x00\x14@'

```

LISTADO 17.12: struct: empaquetado en diferentes tamaños

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 (§5.3).

El valor para los campos de la trama será:

MAC destino

FF:FF:FF:FF:FF:FF (es una trama broadcast).

MAC origen

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

tipo 0x0806, que corresponde con el protocolo ARP.

En el Listado 17.13 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.

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

LISTADO 17.13: struct: empaquetando una cabecera Ethernet

17.7. Desempaquetado

La contrapartida de `pack()` es `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. La función `unpack()` realiza por tanto la *des-serialización*.

El Listado 17.14 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`.

```
1 >>> header
2 b'\xff\xff\xff\xff\xff\xff\xc4\x85\x08\xed\xD3\x07\x08\x06'
3 >>> struct.unpack('!6s6sh', header)
4 (b'\xff\xff\xff\xff\xff\xff', b'\xc4\x85\x08\xed\xD3\x07', 2054)
5 >>> hex(2054)
6 '0x0806'
```

LISTADO 17.14: struct: desempaquetando una cabecera Ethernet

17.8. Formatos de serialización binaria

El módulo `struct` permite trabajar con datos binarios de un modo muy sencillo, pero es bastante limitado. Si los datos a serializar son de tamaño variable, como cadenas o secuencias de objetos, la cosa se complica bastante. Para usos más avanzados, sobre todo cuando queremos definir nuestras propias estructuras, existen librerías mucho más potentes como `construct`, Google Protocol Buffers o Apache Avro.

17.9. Serialización textual

En las capas inferiores —en realidad desde transporte hacia abajo— la gran mayoría de los protocolos utilizan serialización binaria porque normalmente eso genera mensajes más compactos y su procesamiento es más rápido y requiere menos recursos. En la capa de aplicación sin embargo, es común encontrar muchos protocolos que utilizan serialización textual.

El formato textual para codificación de datos mas usado en la actualidad con mucha diferencia es JSON. Como su nombre indica, JSON genera/reconoce cadenas de texto que replican el formato de la definición de objetos con la sintaxis de JavaScript⁹. El grán éxito de JSON se debe a que es muy legible por humanos y su procesamiento automatizado es muy sencillo, aunque costoso en recursos comparado con los formatos binarios. Hoy en día es una norma estandarizada y muchos lenguajes ofrecen soporte incluso en su librería estándar.

Desde Python, también resulta muy sencillo de manipular. El Listado 17.15 muestra como serializar la hipotética lectura de un sensor de temperatura y humedad, que incluye además un identificar y el estado de carga de su batería. El resultado de la serialización (payload) se puede enviar perfectamente como carga útil de un datagrama UDP.

```
>>> import json
>>> import socket

>>> data = {'sensor-id': 'temp-hum-01', 'temperature': 22.5, 'humidity': 45.2,
...         'battery': 0.85}
>>> payload = json.dumps(data)
>>> print(payload)
{"sensor-id": "temp-hum-01", "temperature": 22.5, "humidity": 45.2, "battery": 0.85}
>>> print(type(payload))
<class 'str'>

>>> with socket.socket(socket.AF_INET, socket.SOCK_DGRAM) as sock:
...     sock.sendto(payload.encode(), ('203.0.113.2', 55000))
```

LISTADO 17.15: Serialización JSON y envío de la lectura de un sensor

El código con la deserialización (Listado 17.16) muestra la recepción del mensaje su des-serIALIZACIÓN.

```
>>> import json
>>> import socket

>>> with socket.socket(socket.AF_INET, socket.SOCK_DGRAM) as sock:
...     sock.bind(('', 55000))
...     data, addr = sock.recvfrom(1024)
```

⁹Hay mucho que puntualizar en esta frase, pero sirve como aproximación.

```
>>> data = json.loads(data.decode())
>>> print(data['temperature'])
22.5
```

LISTADO 17.16: Deserialización JSON de la lectura de un sensor

También para serialización textual hay innumerables formatos entre los que podemos destacar XML, YAML, CSV o TOML entre otros.

Y ¿qué más?

La serialización es una parte esencial de las comunicaciones en Internet, pero también una fuente inagotable de errores y confusión sobre todo entre programadores noveles. El objetivo de este capítulo ha sido aclarar los conceptos más básicos mostrando ejemplos prácticos que ilustren su funcionamiento. Quedan fuera muchos detalles técnicos, pero a partir de este punto el lector tendrá la base para profundizar por su cuenta en librerías y formatos con los que se encuentre en su día a día, que a buen seguro, serán muchos y variados.

Capítulo 18

Captura y análisis

Al terminar este capítulo, entenderás:

- Algunas de las opciones que `tshark` y `wireshark` ofrecen para captura y análisis de tráfico de red.
- Qué son los filtros de captura y cómo se usan.
- Qué son los filtros de visualización y qué posibilidades ofrecen.
- Cómo generar estadísticas y resultados a medida.
- Cómo recuperar la carga útil de un flujo UDP o conexión TCP.

Ya has realizado muchas capturas de tráfico con `tshark` desde los primeros capítulos. Pero todas esas veces ha sido con un uso muy básico del programa, `tshark` es capaz de mucho más: puede diseccionar mensajes, interpretar el significado de cada campo, mostrar la información de diferentes formas, generar estadísticas... entre otras cosas. En este capítulo verás de qué más es capaz `tshark`, y su hermano mayor `wireshark`, una versión con interfaz gráfica más potente y versátil.

Este tipo de programas se llaman «analizadores de protocolos» o *sniffers*, y resultan extremadamente útiles para detectar problemas en la red, optimizar protocolos y depurar aplicaciones. Permiten capturar tráfico de una interfaz de red específica o de todas, visualizarlo en tiempo real, o almacenarlo en un archivo para poder analizarlo después, y estudiar capturas que han hecho otros.

Además, permiten aplicar filtros de captura para procesar únicamente los mensajes que coinciden con el criterio que te interese, y también filtros de visualización, para elegir qué mensajes de la captura se muestran en cada momento. Es importante entender esta diferencia. El filtro de captura es permanente, los mensajes descartados por el filtro pierden y no formarán parte de la captura. El filtro de visualización, en cambio, se puede aplicar, modificar o eliminar en cualquier momento y no afecta a la captura, solo a lo que se muestra.

18.1. tshark

A partir de las múltiples capturas que hemos realizado con **tshark** seguro que ya tienes bastante clara su utilidad. **tshark** es un analizador similar al veterano **tcpdump**, aunque con mucha más funcionalidad y posibilidades. Retomemos una captura sencilla de tráfico ICMP como las que ya has visto:

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

```
$ sudo tshark -i eno1 -f icmp
Capturing on eno1
192.168.0.37 -> 64.170.98.30 ICMP 98 Echo (ping) request id=0x405d, seq=10/2560, ttl=64
64.170.98.30 -> 192.168.0.37 ICMP 98 Echo (ping) reply id=0x405d, seq=10/2560, ttl=60
192.168.0.37 -> 64.170.98.30 ICMP 98 Echo (ping) request id=0x405d, seq=11/2816, ttl=64
64.170.98.30 -> 192.168.0.37 ICMP 98 Echo (ping) reply id=0x405d, seq=11/2816, ttl=60
192.168.0.37 -> 64.170.98.30 ICMP 98 Echo (ping) request id=0x405d, seq=12/3072, ttl=64
64.170.98.30 -> 192.168.0.37 ICMP 98 Echo (ping) reply id=0x405d, seq=12/3072, ttl=60
6 packets captured
```

LISTADO 18.1: tshark capturando tráfico ICMP

Como muchos otros programas de consola habituales en los sistemas POSIX, **tshark** acepta una gran variedad de opciones y argumentos que permiten modificar su comportamiento. En el ejemplo anterior, el significado de sus argumentos es el siguiente:

- i **eth0** Capturar tráfico de la interfaz **eth0**.
- f **icmp** Capturar solo mensajes ICMP y descartar el resto.
- c **6** Capturar únicamente 6 paquetes y terminar.

La información de salida está organizada en columnas:

1. Instante de captura, relativo al primer mensaje¹.
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.

La información que se muestra depende del tipo de mensaje capturado. Por ejemplo, si capturas tráfico ARP verás algo muy distinto después de la columna de tamaño:

¹el instante de captura se ha omitido en este ejemplo.


```
$ sudo tshark -i eno1 -f arp -N m
Capturing on eno1
SamsungE_db:d6:45 → Micro-St_30:69:7b ARP 60 Who has 192.168.1.121? Tell 192.168.1.193
Micro-St_30:69:7b → SamsungE_db:d6:45 ARP 42 192.168.1.121 is at 2c:f0:5d:30:69:7b
AzureWav_77:e6:c4 → Micro-St_30:69:7b ARP 60 Who has 192.168.1.121? Tell 192.168.1.181
Micro-St_30:69:7b → AzureWav_77:e6:c4 ARP 42 192.168.1.121 is at 2c:f0:5d:30:69:7b
```

Mediante los argumentos es posible afinar la captura con un alto grado de detalle. Además de capturar tráfico de una interfaz de red específica e imprimir un resumen como el anterior, `tshark` también puede cargar una captura desde un archivo (opción `-r`) y, por supuesto, también crear este tipo de archivos para su análisis posterior (opción `-w`).

18.1.1. Acceso privilegiado

Quizá hayas advertido el uso de `sudo` en los comandos anteriores. Esto se debe a que para capturar tráfico «crudo» de la interfaz de red se requieren privilegios especiales. Puedes utilizar `sudo` para ejecutar un programa con los privilegios de otro usuario (por defecto de `root`). Sin embargo, ejecutar cualquier programa con privilegios de administrador es siempre un riesgo de seguridad, de hecho, así lo advierte el propio `tshark` en su salida.

Existe una alternativa más conveniente para ejecutar `tshark` o programas similares. El núcleo Linux permite asignar «capacidades» (*capabilities*) específicas a un programa (un archivo binario). Las capacidades necesarias para capturar tráfico son `CAP_NET_ADMIN` y `CAP_NET_RAW`. Para aplicarlas al programa que realiza la captura de tráfico para `tshark` (`dumpcap`) basta ejecutar:

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

Esta solución otorga esas capacidades a cualquier usuario que ejecute el programa `dumpcap`. Una alternativa más conservadora es otorgarlas sólo a los miembros de un grupo de usuarios específico (*p. ej.* grupo `wireshark`)².

18.1.2. Selección de la interfaz de captura

Como en el ejemplo anterior, se puede elegir de qué interfaz de red capturar (con la opción `-i`). Si no se especifica esta opción, `tshark` elegirá la interfaz que se esté utilizando para conectar a Internet. Recuerda que para obtener una lista de interfaces puedes utilizar el comando `ip link` o el veterano `ifconfig`. También el propio `tshark` puede mostrar una lista de interfaces disponibles, que además incluye algunas pseudo-interfaces que él mismo define y que en realidad no existen en el sistema.

²Consulta <http://wiki.wireshark.org/CaptureSetup/CapturePrivileges> para más detalles

```
$ tshark -D
1. eno1
2. any
3. lo (Loopback)
4. bluetooth-monitor
5. nflog
6. nfqueue
7. dbus-system
8. dbus-session
9. veth2728659
10. veth1362a83
11. br-6018fd38a646
12. br-7e03d0bfa152
13. ciscodump (Cisco remote capture)
14. randpkt (Random packet generator)
15. sdjournal (systemd Journal Export)
16. sshdump (SSH remote capture)
17. udpdump (UDP Listener remote capture)
```

Demostro un repaso rápido a las interfaces que ha encontrado en este caso. Es probable que muchas de ellas también aparezcan cuando ejecute este comando en tu propio sistema.

eno1

La primera NIC Ethernet, y única en este caso.

any Interfaz virtual que permite capturar tráfico de todas las interfaces.

lo La interfaz *loopback*.

bluetooth-monitor

Captura de tráfico Bluetooth.

nflog y nfqueue

Captura de paquetes de Netfilter, el sistema de cortafuegos de Linux.

dbus-system y dbus-session

Interfaz de captura de mensajes del bus de mensajes local (D-BUS) que utilizan muchos sistemas GNU/Linux.

veth-<id>y br-<id>

Interfaces Ethernet virtuales y *bridge* respectivamente, comúnmente utilizadas por aplicaciones de virtualización como Virtualbox, VMWare, KVM, docker, etc. para conexión a red de máquinas virtuales.

ciscodump

Captura remota de Cisco.

spjournal

Captura de mensajes del sistema systemd.

randpkt

Generación de paquetes aleatorios.

sshdump

Captura remota basado en un túnel SSH.

udpump

Captura remota de paquetes basado en un flujo UDP.

18.1.3. Limitando la captura

Si no se le indica lo contrario, `tshark` continuará capturando tráfico indefinidamente, ya sea mostrando la salida en consola o almacenándolo en un archivo.

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 archivo de la captura resultante, indicando en KiB.

-a files:N

un número de archivos.

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

```
$ tshark -i wlan0 -w wlan0.pcap -a filesize:1024
Capturing on 'wlan0'
529
```

Durante el proceso, `tshark` mostrará en consola la cantidad actual de paquetes capturados.

18.1.4. Filtros de captura

Como hemos visto, el argumento `-f` sirve para especificar un filtro de captura. El formato de estos filtros constituye todo un lenguaje en sí mismo. La tabla 18.1 contiene unos cuantos ejemplos extraídos de <http://wiki.wireshark.org/CaptureFilters>. Puedes encontrar información detallada en la página de manual de `pcap-filter`.

18.1.5. Formato de salida

Por defecto el programa muestra una línea que resume cada mensaje, como en los ejemplos anteriores. Pero existen otras muchas posibilidades para obtener información detallada de cada campo. La opción `-v` muestra los valores de los campos de cada mensaje (por capa) en un formato de texto

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

CUADRO 18.1: tshark: ejemplos de filtros de captura

legible (Listado 18.2). Ya vimos una parte de esto en la captura 5.1 dónde introdujimos brevemente la información que ofrece. Aparte de los valores directos que se obtienen al diseccionar las cabeceras, *tshark* también muestra información derivada o calculada que no aparece en los mensajes, como por ejemplo, los intervalos de tiempo entre mensajes o el resultado de la comprobación de los checksums. Esa información aparece entre corchetes.

```
$ tshark -f icmp -c 1 -v
Frame 1: 98 bytes on wire (784 bits), 98 bytes captured (784 bits) on interface 0
  Interface id: 0
    WTAP_ENCAP: 1
    Arrival Time: Feb 12, 2013 16:59:13.856520000 CET
    [Time shift for this packet: 0.000000000 seconds]
    Epoch Time: 1360684753.856520000 seconds
    [Time delta from previous captured frame: 0.000000000 seconds]
    [Time delta from previous displayed frame: 0.000000000 seconds]
    [Time since reference or first frame: 0.000000000 seconds]
    Frame Number: 1
    Frame Length: 98 bytes (784 bits)
    Capture Length: 98 bytes (784 bits)
    [Frame is marked: False]
    [Frame is ignored: False]
    [Protocols in frame: eth:ip:icmp:data]
  Ethernet II, Src: Intel_ef:d4:96, Dst: Cisco_3a:c9:40
    Destination: Cisco_3a:c9:40 (00:64:40:3a:c9:40)
      Address: Cisco_3a:c9:40 (00:64:40:3a:c9:40)
      .... ..0. .... = LG bit: Globally unique address (factory default)
      .... ..0. .... = IG bit: Individual address (unicast)
    Source: Intel_ef:d4:96 (c4:85:08:ef:d4:96)
      Address: Intel_ef:d4:96 (c4:85:08:ef:d4:96)
      .... ..0. .... = LG bit: Globally unique address (factory default)
      .... ..0. .... = IG bit: Individual address (unicast)
    Type: IP (0x0800)
```

```

Internet Protocol Version 4, Src: 120.12.17.214, Dst: 12.22.58.30
  Version: 4
  Header length: 20 bytes
  Differentiated Services Field: 0x00 (DSCP 0x00: Default; ECN: 0x00: Not-ECT (Not
  ↳ ECN-Capable Transport))
  Total Length: 84
  Identification: 0x0000 (0)
  Flags: 0x02 (Don't Fragment)
    0... .... = Reserved bit: Not set
    .1.. .... = Don't fragment: Set
    ..0. .... = More fragments: Not set
  Fragment offset: 0
  Time to live: 64
  Protocol: ICMP (1)
  Header checksum: 0x415c [correct]
  Source: 120.12.17.214 (120.12.17.214)
  Destination: 12.22.58.30 (12.22.58.30)
  [Source GeoIP: Unknown]
  [Destination GeoIP: Unknown]
Internet Control Message Protocol
  Type: 8 (Echo (ping) request)
  Code: 0
  Checksum: 0x1fb8 [correct]
  Identifier (BE): 27502 (0x6b6e)
  Identifier (LE): 28267 (0x6e6b)
  Sequence number (BE): 10557 (0x293d)
  Sequence number (LE): 15657 (0x3d29)
  Timestamp from icmp data: Feb 12, 2013 16:59:13.000000000 CET
  [Timestamp from icmp data (relative): 0.856520000 seconds]
  Data (48 bytes)

0000  8c 11 0d 00 00 00 00 00 10 11 12 13 14 15 16 17  .....
0010  18 19 1a 1b 1c 1d 1e 1f 20 21 22 23 24 25 26 27  ..... !"#%&'
0020  28 29 2a 2b 2c 2d 2e 2f 30 31 32 33 34 35 36 37  ()*+,-./01234567
      Data: 8c110d0000000000101112131415161718191a1b1c1d1e1f...
      [Length: 48]

```

LISTADO 18.2: Modo «verboso» de tshark

Normalmente esto es demasiada información y lo interesante es aislar específicamente los campos relacionados con la cuestión que te interese en cada caso. Para lograrlo se puede indicar el formato de salida por campos (-T fields) y la lista con los campos concretos, que pueden depender de las condiciones de filtrado. El ejemplo del Listado 18.3 filtra los mensajes TCP dirigidos al puerto 80 (presuntamente HTTP) y muestra únicamente las direcciones MAC e IP origen y destino.

```

$ tshark -f "tcp and dst port 80" -T fields -e eth.src -e ip.src -e eth.dst -e ip.dst
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
```

LISTADO 18.3: tshark permite elegir los campos a mostrar

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

Además, es posible generar una representación de la captura en otros formatos, como PostScript XML, por si necesitas hacer un tratamiento automatizado de los valores capturados.

18.1.6. Filtros de visualización

A partir de la captura obtenida (o cargada desde archivo) puedes aplicar un filtro que determina qué mensajes se muestran, es decir, un filtro de visualización (*display filter*) altera únicamente lo que el usuario puede ver en cada momento, pero a diferencia de los filtros de captura, estos se pueden editar obteniendo un subconjunto diferente de entre los mensajes capturados. El formato de estos filtros es distinto al de los filtros de captura, y mucho más potente.

Como ejemplo, el siguiente listado muestra las peticiones HTTP que soliciten una URI que contenga la cadena 'favicon.ico', es decir, el icono que los navegadores asocian al guardar una página como favorito (*bookmark*).

```
$ tshark -R 'http.request.uri contains "favicon.ico"'
Capturing on wlan0
192.168.0.37 -> 93.184.220.111 HTTP 380 GET /i/favicon.ico?m=1317424629g HTTP/1.1
192.168.0.37 -> 217.148.71.165 HTTP 365 GET /favicon.ico HTTP/1.1
192.168.0.37 -> 23.37.161.157 HTTP 633 GET /favicon.ico HTTP/1.1
11 packets dropped
3 packets captured
```

El programa proporciona literalmente miles de filtros. Puedes consultarlos en la referencia en línea³, en la página de manual para `wireshark-filter`⁴ o con el comando `tshark -G fields`. Estos filtros están organizados jerárquicamente siendo el primer componente el nombre del protocolo al que aplican. En el ejemplo anterior, `http.request.uri` se refiere a la URI que aparece en la petición (GET) de los mensajes HTTP⁵. La tabla 18.2 muestra algunos ejemplos representativos similares a `http://wiki.wireshark.org/DisplayFilters`.

FiXme Fatal: : URL chuleta, copiar en el repo del libro? (ver licencia)

FiXme
Fatal!

³<http://www.wireshark.org/docs/dfref/>

⁴`man wireshark-filter`

⁵<http://www.wireshark.org/docs/dfref/t/tcp.html>

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

CUADRO 18.2: tshark: ejemplos de filtros de visualización

Al igual que los filtros de captura, también es posible utilizar operadores lógicos y relacionales que permiten escribir expresiones muy elaboradas⁷.

chuleta

Filtros de visualización⁶

18.1.7. Estadísticas

tshark genera una gran variedad de estadísticas útiles⁸. En esta sección se incluyen solo algunas a modo de ejemplo.

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 -> 192.168.0.37 HTTP 245 HTTP/1.1 200 OK
(text/plain) tcp.srcport == 80
8.928316 173.194.78.125 -> 192.168.0.37 TCP 471
[TCP segment of a reassembled PDU] tcp.srcport == 5222
17.660298 173.194.78.125 -> 192.168.0.37 TCP 199
[TCP segment of a reassembled PDU] tcp.srcport == 5222
```

icmp,srt[,filter]

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

```
$ tshark -z icmp,srt
[...]
10024 packets captured
ICMP Service Response Time (SRT) Statistics (all times in ms):
Filter: <none>
Requests Replies Lost % Loss
1784 1783 1 0,1%
Minimum Maximum Mean Median SDeviation Min Frame Max Frame
```

⁷http://www.wireshark.org/docs/wsug_html_chunked/ChWorkBuildDisplayFilterSection.html

⁸Puedes ver todas las estadísticas disponibles con `tshark-z help`

55,228 818,109 62,742 56,894 37,105 7831 5373

io,phs[,filter]

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

```
$ tshark -z io,phs -q -c 100
Capturing on wlan0
100 packets captured
Protocol Hierarchy Statistics
Filter:
eth                                frames:100 bytes:18503
ip                                 frames:98 bytes:18289
icmp                              frames:32 bytes:3136
udp                               frames:34 bytes:3760
  dns                             frames:32 bytes:3408
  db-lsp-disc                      frames:2 bytes:352
tcp                               frames:32 bytes:11393
  ssl                             frames:14 bytes:10189
    tcp.segments                  frames:2 bytes:1157
llc                               frames:2 bytes:214
data                             frames:2 bytes:214
```

io,stat,interval,filter,filter

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

```
$ 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 |
|-----|
=====
```


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 |
|-----|
```

follow,proto,mode,filter[,range]

Muestra el contenido de un flujo de mensajes entre dos nodos, 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 archivo podría recuperarse a partir de la captura. Los protocolos soportados son TCP y UDP. Y puede hacer el volcado en tres modos diferentes: ascii, hex y raw.

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

conv,type[,filter]

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

```
$ tshark -z conv,tcp
TCP Conversations
```

Filter:<No Filter>		<-		->		Total		Start	Durat
		F	B	F	B	F	B		
192.168.0.37:38216	<-> 73.233.104.123:80	1	74	2	140	3	214	0,002912	0,2126
192.168.0.37:58949	<-> 92.184.220.111:80	1	66	2	128	3	194	0,002801	0,1134
192.168.0.37:58948	<-> 92.184.220.111:80	1	66	2	128	3	194	0,002754	0,1118
192.168.0.37:58947	<-> 92.184.220.111:80	1	66	2	128	3	194	0,002704	0,1027
192.168.0.37:58946	<-> 92.184.220.111:80	1	66	2	128	3	194	0,002660	0,1015
192.168.0.37:56569	<-> 92.100.127.144:80	1	74	2	140	3	214	0,002398	0,0727
192.168.0.37:38838	<-> 95.172.94.11:80	1	62	2	128	3	190	0,002319	0,0952
192.168.0.37:38837	<-> 95.172.94.11:80	1	62	2	128	3	190	0,002270	0,0935

18.2. wireshark

wireshark es una herramienta gráfica que amplía las posibilidades de tshark. Permite visualizar el tráfico capturado de un modo más flexible, potente e intuitivo, y además ofrece herramientas de análisis más sofisticadas.

18.2.1. Interfaz gráfica

Al arrancar wireshark aparece una ventana (Figura 18.1) que muestra la lista de interfaces disponibles para captura (las mismas que hemos visto con tshark en § 18.1.2). Desde aquí puedes especificar un filtro de captura (opcional), seleccionar una interfaz y pulsar el botón de captura: .

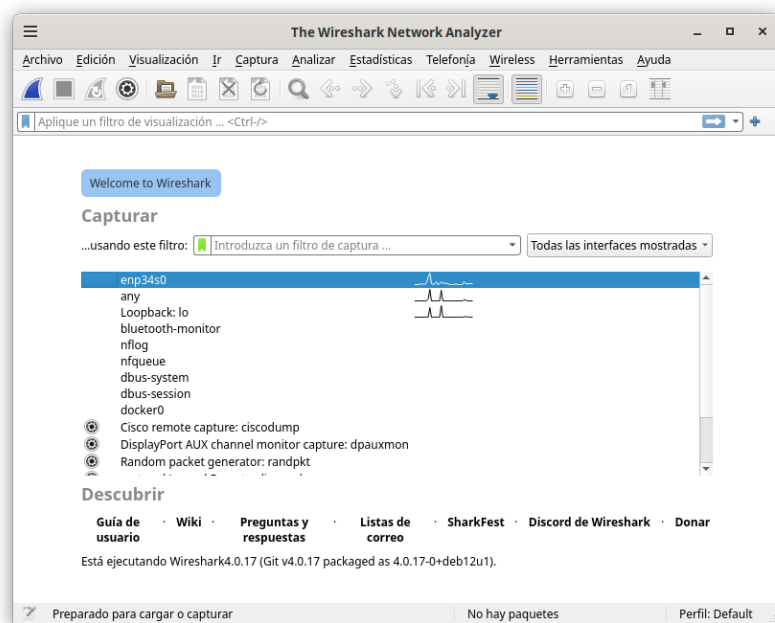


FIGURA 18.1: wireshark: interfaz inicial

Inmediatamente la ventana muestra 3 paneles:

1. El panel superior muestra la lista de mensajes capturados, con una línea por mensaje, que se va actualizando en tiempo real conforme van llegando nuevos mensajes. Se muestran por defecto las columnas No., Time, Source, Destination, Protocol, Length e Info.
2. Abajo a la izquierda, aparece un árbol expandible que incluye todos los detalles de los campos de cada una de las cabeceras de los distintos protocolos encapsulados en cada mensaje.
3. Abajo a la derecha, el contenido completo del mensaje en formato hexadecimal en un formato prácticamente idéntico al que hemos visto con tshark al indicar la opción `-v` en § 5.1.

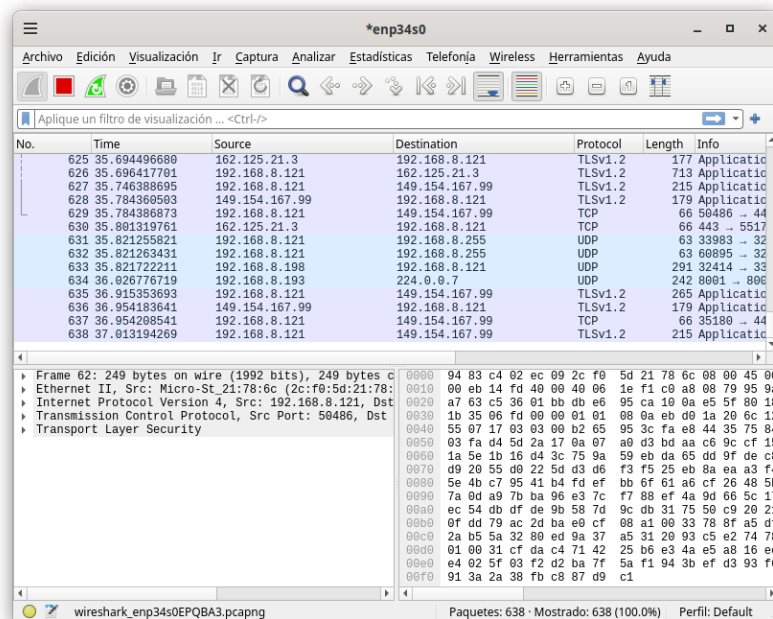


FIGURA 18.2: wireshark: interfaz de captura por defecto

Esta vista por defecto no parece la más útil. Puedes cambiar entre distintas disposiciones en el menú Edición→Preferencias→Apariencia→Diseño y elegir el diseño de 3 filas (el primero) que corresponde con lista de mensajes, detalles y contenido (bytes) tal como muestra la Figura 18.3. De ese modo la interfaz se verá como en la Figura 18.4.

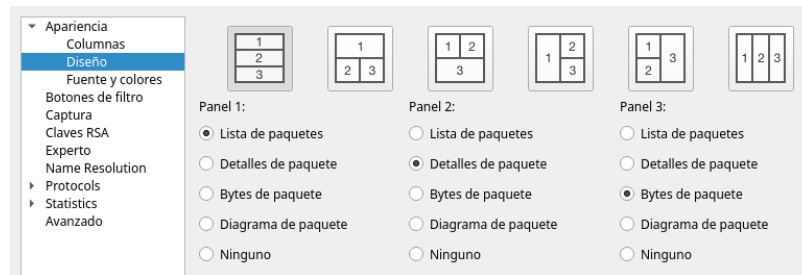


FIGURA 18.3: wireshark: configuración de paneles

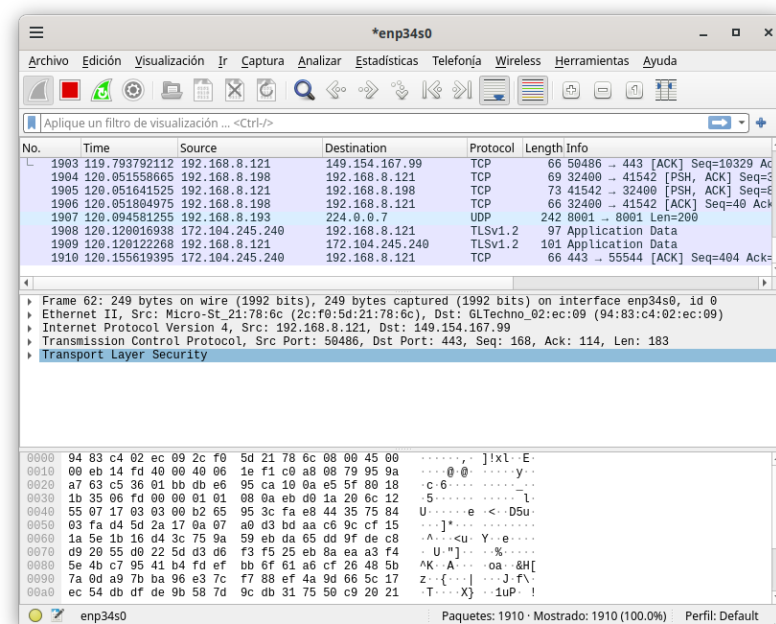


FIGURA 18.4: wireshark: interfaz con las 3 filas: lista, detalle y bytes

Habrás comprobado que el programa sigue capturando tráfico indefinidamente, hasta que pulses el botón de parada: .

18.2.2. Captura y filtrado

Como ejemplo, vamos a repetir la captura que hicimos con `tshark` en § 5.9.2 para ilustrar una conexión TCP. En la ventana inicial escribe `tcp port 2000` como filtro de captura, selecciona la interfaz *loopback* y lanza la captura (Figura 18.5).

Comprobarás que no aparece nada... es normal, por el momento no hay tráfico que coincida con el filtro de captura. Vamos a generarlo recreando la misma sesión TCP con `ncat`, es decir, en una consola ejecuta el servidor con `ncat -l -p 2000` y en otra el cliente con `echo hola | ncat 127.0.0.1 2000`. Inmediatamente verás que aparecen 8 mensajes en el panel superior, como en la Figura 18.6.

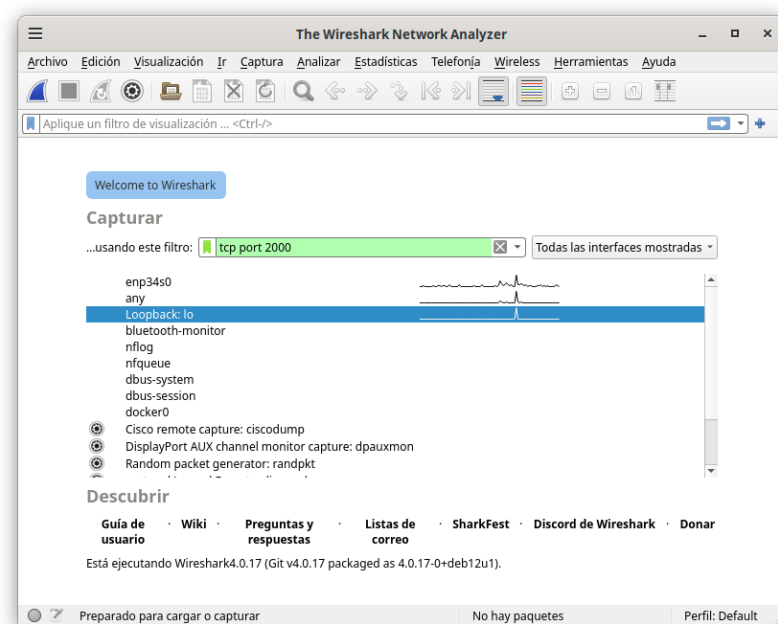


FIGURA 18.5: wireshark: estableciendo una captura en la interfaz *loopback* para el puerto TCP 2000

El primero de los mensajes está seleccionado y eso permite explorar todos sus campos en el panel central. En la figura, la parte correspondiente a la cabecera TCP se muestra expandida y aparece el valor de cada campo. En esta segunda sección, el campo seleccionado es «flags» con el valor `0x002`, que corresponde a la activación del flag SYN. Efectivamente se trata del segmento de establecimiento de conexión enviado por el cliente. A su vez, esto provoca que los bytes del mensaje que corresponden a ese campo de la cabecera aparezcan resaltados en el panel inferior.

Aquí puedes establecer un filtro de visualización para, por ejemplo, quedarte únicamente con los segmentos TCP que tienen carga útil. Ese filtro será `tcp.len > 0` y como puedes ver en Figura 18.7 se escribe en la barra que queda resaltada con fondo verde (ese color confirma que el filtro es correcto).

El único mensaje que cumple el filtro es el que lleva la cadena «hola\n», y lo puedes ver resaltado en los paneles central e inferior.

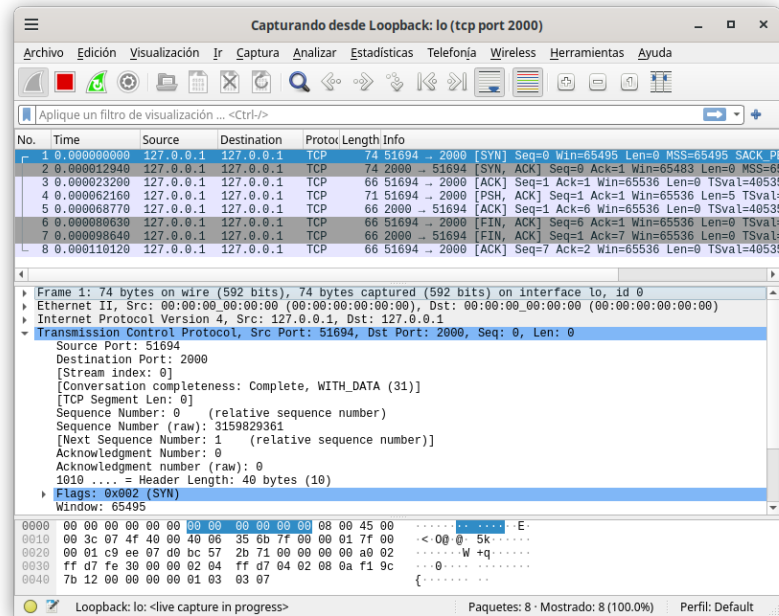


FIGURA 18.6: wireshark: captura de una conexión TCP simple

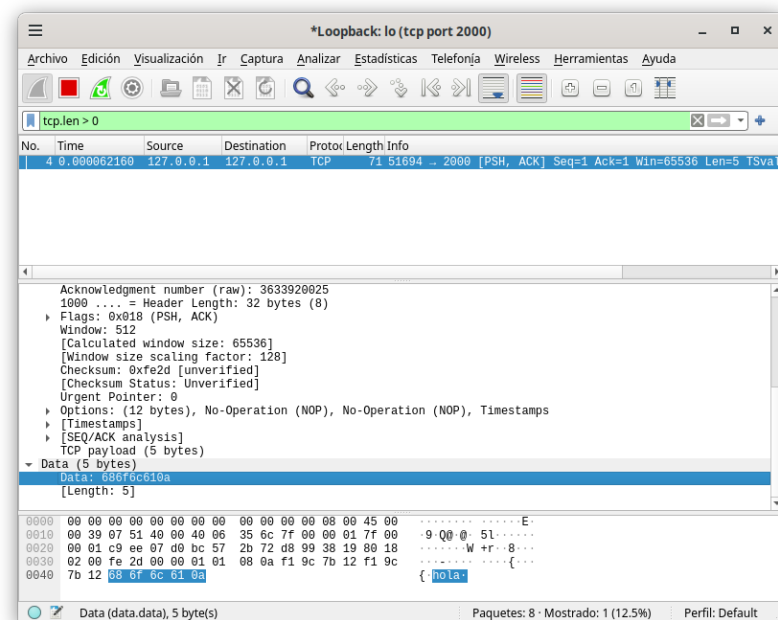


FIGURA 18.7: wireshark: filtro de visualización (mensajes con carga útil)

18.3. Captura de flujos

wireshark ofrece la posibilidad de capturar la carga útil de una conexión TCP o de un flujo UDP en uno o en ambos sentidos. Esto significa que, por ejemplo, es posible recuperar y reconstruir un archivo que se está transmitiendo a través de la red, asumiendo, por supuesto, que has sido capaz de capturar todos los mensajes.

Para empezar debes utilizar tshark para capturar el tráfico dirigido al puerto TCP 2000 y lo puedes guardar en un archivo. Después analizaremos ese archivo para recuperar la carga útil de la conexión completa tanto con wireshark como con herramientas de línea de comandos. Por supuesto, con tshark por ejemplo, aunque por supuesto lo puedes hacer igualmente con wireshark.

```
$ tshark -i lo -f "tcp port 2000" -w captura.pcap
```

Ahora envía un archivo (un .mp3 concretamente, aunque sirve cualquier cosa) a través de una conexión TCP que puedes conseguir con un servidor ncat a la escucha en el puerto 2000 y un cliente (también ncat) con el que enviar el archivo:

Cliente	Servidor
\$ cat original.mp3 nc -l localhost 2000	\$ nc -l -p 2000 > recibido.mp3

FIGURA 18.8: Transfiriendo un .mp3 con nc

Una vez terminado el envío, puedes parar la captura con **Ctrl+C**. Y la puedes cargar en **Wireshark** con la opción **Archivo→Abrir** o desde consola simplemente con:

```
$ wireshark captura.pcap
```

Ahora verás la lista de todos los mensajes que forman la captura. Puedes abrir el menú contextual pulsando con el botón derecho del ratón en cualquier de los mensajes. Elige **Seguir→TCP Stream**. Aparecerá una nueva ventana como la de la Figura 18.9.

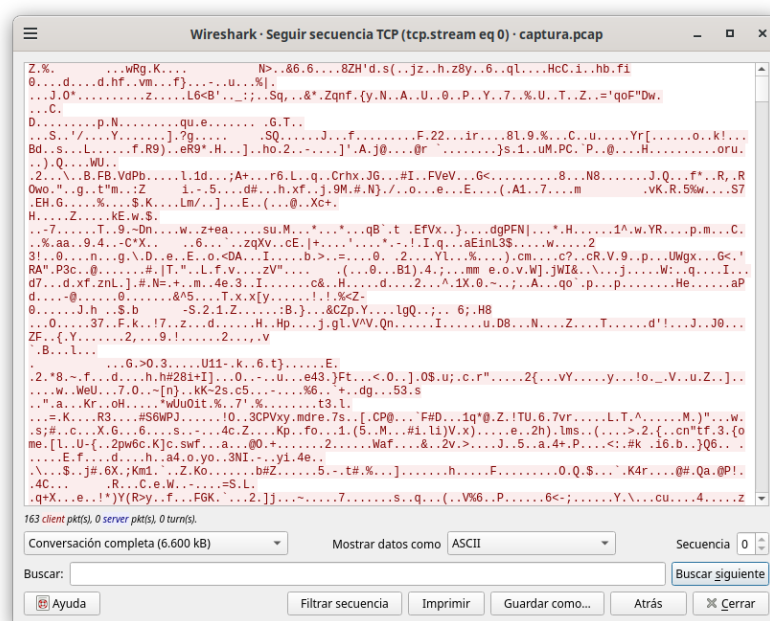


FIGURA 18.9: wireshark: reconstrucción de un flujo TCP

En los controles que aparecen en la parte inferior elige «Mostrar como: raw» y pulsa el botón «Guardar como...». Guarda el archivo con el nombre **captura.mp3**.

Por supuesto, puedes abrirlo con cualquier reproductor multimedia, pero también puedes comprobar que el archivo recuperado es idéntico al original utilizando el comando `diff` o calculando el MD5 de ambos archivos.

```
$ md5sum *  
c2b91faddfec1d01c4f81a8a6d34d537 original.mp3  
c2b91faddfec1d01c4f81a8a6d34d537 captura.mp3
```

Puedes conseguir lo mismo desde línea de comandos con:

```
$ tcpflow -r captura.pcap
```

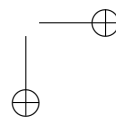
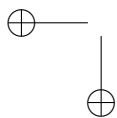
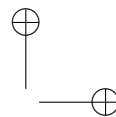
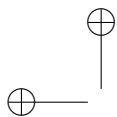
El programa genera un archivo con la carga útil para cada uno de los flujos y sentidos contenidos en la captura. En este caso solo crea un archivo, ya que el sentido servidor a cliente de la conexión TCP no ha transportado ningún dato. El archivo se llama en este caso `127.000.000.001.58008-127.000.000.001.02000` que está formado por las direcciones IP y puertos de origen y destino de la conexión. Igual que antes, puedes comprobar que el archivo es idéntico al original.

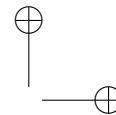
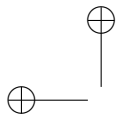
```
$ md5sum *  
c2b91faddfec1d01c4f81a8a6d34d537 original.mp3  
c2b91faddfec1d01c4f81a8a6d34d537 captura.mp3  
c2b91faddfec1d01c4f81a8a6d34d537 127.000.000.001.58008-127.000.000.001.02000
```

Puedes ver que resulta sorprendente sencillo recuperar un archivo completo que se transfiere por la red si se dan las condiciones y se dispone de las herramientas adecuadas. Es fácil entender las consecuencias que esto tiene para la privacidad. Por supuesto, si el archivo se transmite sobre un canal cifrado (como TLS) esto no será posible, o al menos, no tan sencillo.

Y ¿qué más?

En este capítulo hemos dado un vistazo a algunas de las posibilidades de captura, visualización, filtrado, análisis y generación de estadísticas que ofrecen `tshark` y `wireshark`. Echando un vistazo a su documentación es fácil percatarse de que se trata de herramientas extraordinariamente potentes, que se utilizan habitualmente en ámbitos profesionales de la seguridad informática, la administración de redes, el desarrollo de software de comunicaciones, etc. A partir de aquí, invitamos al lector a explorar su gran potencial. Hay mucho que descubrir y aprender.





Capítulo 19

Sockets raw

Los sockets más usados con diferencia son los TCP seguidos de los UDP. Pero hay muchos otros tipos de socket. Este capítulo es una introducción muy práctica a los «sockets raw»¹.

El término *raw* en informática se suele utilizar para indicar acceso directo a los datos que proporciona un dispositivo, o al menos con menor intervención del SO o librerías involucradas². 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.

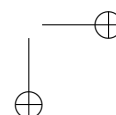
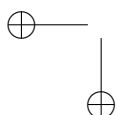
Estas tres cuestiones se pueden aplicar casi a cualquier dispositivo que permita un acceso «raw», sea un periférico USB, una consola o, como en este caso, un socket.

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

¹A veces (dolorosamente) traducido como «conector directo».

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

³a menos que acudamos a llamadas al sistema como `setsockopt()`^{sc}.



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 `ping`, `tracert`, `arping` o un *sniffer* cualquiera. Entonces ¿cómo se hacen estos programas? La respuesta, como habrás imaginado, pasa por los sockets raw.

19.1. Acceso privilegiado

En §18.1.1 hablamos de la necesidad de `sudo` o *capabilities* para ejecutar un sniffer y eso se debe precisamente a que utilizan sockets raw, de modo que todo eso es aplicable a cualquier programa que escribamos y también los utilice.

Lamentablemente, las *capabilities* solo se pueden aplicar a programas binarios, no a scripts de Python. Por eso, para ejecutar los programas que veremos en este capítulo, tendrás que utilizar `sudo`.

Aparte de la ejecución del programa, también necesitas configurar la interfaz de red en «modo promiscuo». Si tu interfaz de red es una tarjeta Ethernet o WiFi, 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 lograrlo debes activar el «modo promiscuo» de la NIC, algo que puede hacer con el comando `ip`:

```
$ sudo ip link set eth0 promisc on
```

O con `ifconfig`:

```
$ sudo ifconfig eth0 promisc
```

De forma análoga puedes saber si la interfaz está en modo promiscuo con el siguiente comando (fíjate en el flag `PROMISC`).

```
$ 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
```

```

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

```

19.2. Tipos de sockets raw

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.

19.3. 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.

```

import socket

ETH_P_ALL = 3

sock = socket.socket(socket.AF_PACKET, socket.SOCK_RAW,
                     socket.htons(ETH_P_ALL))

while 1:
    print("--\n{!r}".format(sock.recvfrom(1600)))

```

LISTADO 19.1: Sniffer básico con AF_PACKET:SOCK_RAW
[📄/raw/sniff-all.py](#)

Y a continuación el programa en funcionamiento:

```
$ sudo ./sniff-all.py
--
(b'\xff\xff\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.⁴ 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:

```
import sys
import socket

if len(sys.argv) != 2:
    print("usage: {} <iface>".format(sys.argv[0]))
    exit(1)

ETH_P_ARP = 0x0806

sock = socket.socket(socket.AF_PACKET, socket.SOCK_RAW,
                     socket.htons(ETH_P_ARP))
sock.bind((sys.argv[1], ETH_P_ARP))

while 1:
    print("--\n{!r}".format(sock.recv(1600)))
```

LISTADO 19.2: Sniffer AF_PACKET:SOCK_RAW filtrando ARP
 /raw/sniff-arp.py

Y el programa en acción:

```
$ sudo ./sniff-arp.py wlan0
--
b'\xff\xff\xff\xff\xff\xff\xff\xffm\x84y\x1d\x08\x06\x00\x01\x08\x00\x06\x04\x00\x01m\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\x00\x00'
```

⁴<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, es el tipo para 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 (y riesgos de seguridad).

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.

19.3.1. 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`⁵.

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)

```
import sys
import socket
import struct

if len(sys.argv) != 2:
    print("usage: {} <iface>".format(sys.argv[0]))
    exit(1)

ETH_P_ARP = 0x0806

sock = socket.socket(socket.AF_PACKET, socket.SOCK_RAW,
                     socket.htons(ETH_P_ARP))
sock.bind((sys.argv[1], ETH_P_ARP))

sock.send(struct.pack('!6s6sh', 6 * b'\xFF',
                       b'\x00\x01\x02\x03\x04\x05', ETH_P_ARP))
```

LISTADO 19.3: Enviando una trama Ethernet con AF_PACKET:SOCK_RAW
[🔗/raw/send-wrong-eth.py](#)

⁵Consulte el capítulo 17 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.

19.3.2. 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`.

19.3.2.1. 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:

```
class Ether:
    def __init__(self, hwsrc, hwdst):
        self.hwsrc = hwsrc
        self.hwdst = hwdst
        self.payload = None

    def set_payload(self, payload):
        self.payload = payload
        payload.frame = self

    def serialize(self):
        retval = struct.pack("!6s6sh", self.hwdst, self.hwsrc,
                             self.payload.proto) + self.payload.serialize()

        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.

Esos datos se empaquetan en binario gracias a `struct.pack()`⁶ 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:

```
class ArpRequest:
    proto = ETH_P_ARP

    def __init__(self, psrc, pdst):
        self.psrc = socket.inet_aton(psrc)
        self.pdst = socket.inet_aton(pdst)
        self.frame = None

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

19.3.2.2. 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:

```
class EtherDissector:
    def __init__(self, frame):
        try:
            (self.hwdst,
             self.hwsrc,
             self.proto) = struct.unpack("!6s6sh", frame[:14])
        except struct.error:
            raise DissectionError

        self.payload = frame[14:]
```

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

⁶Ver <http://docs.python.org/library/struct.html#format-characters>

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

```
class ArpReplyDissector:
    def __init__(self, msg):
        self.msg = msg

        if struct.unpack("!H", self.msg[6:8])[0] != ARP_REPLY:
            raise DissectionError

        try:
            (self.hwsrc, self.psrc,
             self.hwdst, self.pdst) = struct.unpack("!6s4s6s4s", msg[8:28])
        except struct.error:
            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`.

19.3.2.3. Programa principal

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

```
def main(ipsrc, ipdst, iface):
    print("Request: Who has {0}? Tell {1}".format(ipdst, ipsrc))

    sock = socket.socket(socket.AF_PACKET, socket.SOCK_RAW,
                        socket.htons(ETH_P_ARP))
    sock.bind((iface, ETH_P_ARP))

    frame = Ether(sock.getsockname()[-1], BROADCAST)
    frame.set_payload(ArpRequest(ipsrc, ipdst))

    sock.send(frame.serialize())

    while 1:
        eth = EtherDissector(sock.recv(2048))

        try:
            arp_reply = ArpReplyDissector(eth.payload)
            if arp_reply.hwdst == frame.hwsrc:
                print("Reply: {0} is at {1}".format(
                    ipdst, display_mac(arp_reply.hwsrc)))
                break
        except DissectionError:
            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 archivo `raw/arping.py`.

19.4. 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.

19.4.1. Capturando mensajes

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

```
import socket

sock = socket.socket(socket.AF_INET, socket.SOCK_RAW,
                     socket.getprotobyname('udp'))

while 1:
    print("--\n{!r}".format(sock.recv(1600)))
```

LISTADO 19.4: Sniffer de mensajes UDP con `AF_INET:SOCK_RAW`
 `/raw/sniff-udp.py`

La función `getprotobyname()` devuelve el número de protocolo⁷ a partir de su nombre (línea 4). Es interesante destacar que el resultado del método `recv()` es el paquete IP completo, incluyendo cabecera (línea 7).

⁷<http://www.iana.org/assignments/protocol-numbers/protocol-numbers.xml>

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).

19.4.2. Enviando

Para enviar datos sobre este tipo de socket debes utilizar el método `sendto()` indicando la dirección destino. El Listado 19.5 envía un segmento UDP «sintético», pero válido, que contiene el texto «hello Inet».

```
import socket
import struct

sock = socket.socket(socket.AF_INET, socket.SOCK_RAW,
                     socket.getprotobyname('udp'))

payload = b'hello Inet'
udp_pkt = struct.pack('!4h', 0, 2000, 8+len(payload), 0) + payload
sock.sendto(udp_pkt, ('127.0.0.1', 0))
```

LISTADO 19.5: Sintetizando un mensaje UDP con `AF_INET:SOCK_RAW`
 `/raw/send-udp.py`

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».

19.4.2.1. 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.

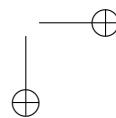
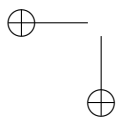
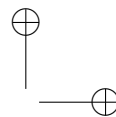
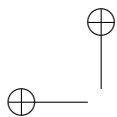
Esta opción, como la gran mayoría, debe fijarse explícitamente después de crear el socket por medio del método `setsockopt()`, tal como se indica:

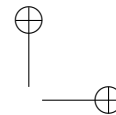
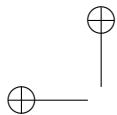
```
sock.setsockopt(socket.SOL_IP, socket.IP_HDRINCL, 1)
```

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.





Capítulo 20

Redes Privadas

Al terminar este capítulo, entenderás:

- Qué es una red privada y por qué es útil.
- Los diferentes tipos de redes privadas: LAN, interredes privadas, redes híbridas, intranets y extranets.
- Qué es el direccionamiento privado y cuáles son bloques de direcciones reservadas.
- Qué el NAT, qué problema resuelve y cuáles provoca.

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.) están bajo el control exclusivo² de dicha organización, y por tanto, también es responsable de su diseño, implementación, gestión y explotación.

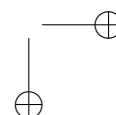
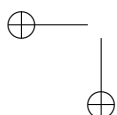
El concepto «red privada» no es equivalente a «red aislada». La red privada más común hoy día es la red WiFi doméstica que encontramos en la mayoría de los hogares y que, obviamente, está conectada a Internet a través del llamado router doméstico y la red del ISP.

20.1. Líneas alquiladas

También una interred puede ser privada, es decir, podemos tener una colección de LAN interconectadas mediante routers, ya sea en una o varias localizaciones, todo eso bajo el control y gestión de una misma organización (eso es lo que la hace privada). Durante los años 80 a 90, cuando Internet no existía o era demasiado caro, era habitual que muchas empresas tuvieran una red privada que conectaba las LAN de sus oficinas (por ejemplo,

¹Habitualmente una LAN

²No necesariamente debe ser su propietario.



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 (muy pocos cientos de metros). Este planteamiento es el que muestra la Figura 20.1.

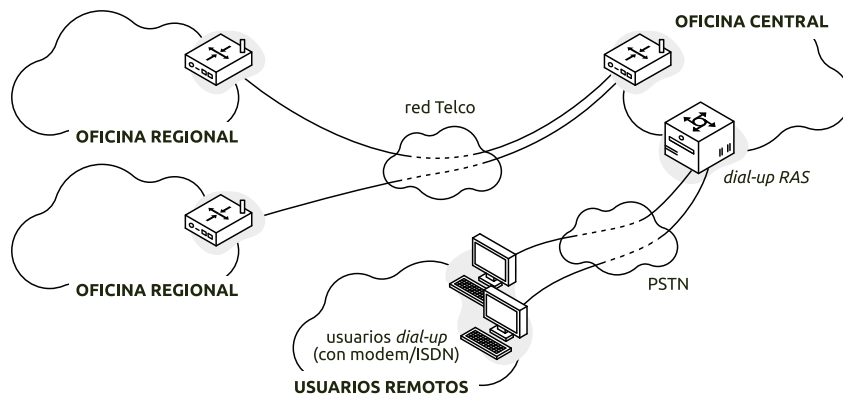


FIGURA 20.1: Red privada que utiliza líneas alquiladas

Cuando partiendo de una topología de este tipo, alguno de las oficinas tiene acceso a Internet puede ofrecer conectividad a algunas o todas las demás oficinas. Este esquema (que se muestra en la Figura 20.2) y es lo que se denomina *red híbrida*.

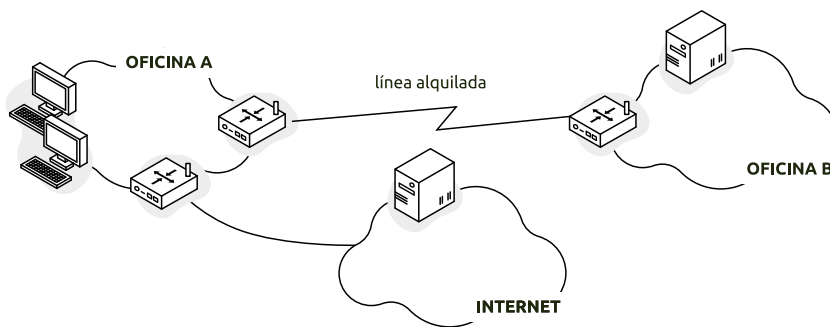


FIGURA 20.2: Red privada híbrida

20.2. 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, parece buena utilizar la pila TCP/IP por variedad y disponibilidad de software, hardware de red, soporte y sobre todo, como forma de simplificar la conexión de la red privada con otras redes. La única razón para no utilizar TCP/IP es que la red privada es que no cumpla con algún requisito técnico específico.



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

Una *intranet* es justamente una red privada que utiliza tecnología TCP/IP y 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 al personal de una organización para lo que requiere autenticación específica.

Como caso particular de *intranet*, la *extranet* permite a ciertos usuarios o servicios acceder a los recursos de la red privada desde el exterior. Este acceso está controlado mediante algún sistema de autenticación y autorización.

20.2.1. Direccionamiento privado

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

- Solicitar un bloque de direcciones públicas globales. Implica realizar una petición, y su correspondiente pago, a las autoridades de Internet: IANA o las entidades regionales FIXME RIR 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 según la RFC 1918 [34] y consiste en la elección arbitraria de uno de los bloques definidos en el Cuadro 20.1. 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 20.1: Bloques IP reservados para direccionamiento privado

Las direcciones privadas deben ser consideradas *no routables* en Internet, 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 privados sin necesidad de autorización. Aunque existan millones de redes alrededor del mundo que estén utilizando exactamente el mismo bloque no hay problema, ya que su tráfico nunca podrá ser confundido con el de otra red. Es decir, las direcciones privadas deben ser localmente únicas, pero al contrario de las públicas, no son globalmente únicas.

Aunque el núcleo de Internet (esencialmente BGP) y los ISP descarten este tráfico, la organización puede encaminarlos dentro de su interred corporativa. Esto le otorga gran flexibilidad a la organización, pudiendo crear varias subredes interconectadas para diferentes propósitos o comunidades de usuarios.

20.3. Conectividad en redes privadas

Las redes domésticas actuales, y la mayoría de las que se utilizan en empresas y organizaciones pequeñas y medianas, técnicamente son *redes privadas híbridas con tecnología TCP/IP*. En una red doméstica, el ISP proporciona un dispositivo (ver Figura 20.3), que llamamos informalmente como *router doméstico*. Este router es el que proporciona la conexión de tus dispositi-

vos a Internet. En realidad ese dispositivo (esa caja) es más que un router. Incorpora normalmente varios dispositivos y funciones:

- Un router IP con 2 interfaces: LAN y WAN.
- Un conmutador Ethernet.
- Un punto de acceso WiFi.
- Un módem que puede utilizar distintas tecnologías: ADSL, SDSL o actualmente fibra óptica con FTTH, HFC y otras. En las conexiones de fibra, el módem puede ser un dispositivo aparte llamado ONT o estar integrado también en la misma caja.
- Un pequeño computador habitualmente con una alguna variante de Linux empotrado.
- Un servidor DHCP.

En la figura puedes ver las conexiones del router doméstico. El primer conector de la derecha (WAN) conecta el equipo con el ONT. Los conectores amarillos numerados de 1 a 4 son los puertos del conmutador Ethernet incorporado. Y la antena, es parte del punto de acceso WiFi.

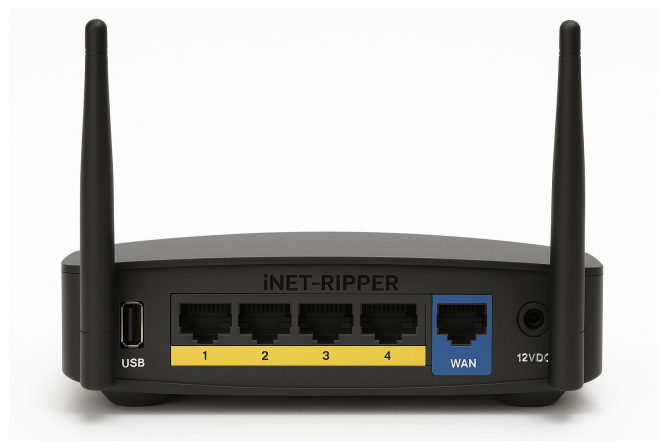


FIGURA 20.3: Conexiones de un router doméstico

Cuando conectas tu computador, móvil, televisor, etc. —sea con Ethernet o WiFi— el servidor DHCP incorporado le asigna una dirección privada. El bloque de direcciones configurado suele ser `192.168.0.0/24`, aunque puede ser cualquiera del Cuadro 20.1, y normalmente el propio usuario tiene la posibilidad de cambiarlo accediendo a la web de administración del router.

Si como hemos dicho, la Internet pública descarta los paquetes IP dirigidos a direcciones privadas ¿cómo pueden entonces estos nodos comunicarse con

servidores públicos? La solución consiste en *traducir* las direcciones privadas a direcciones públicas al salir de la red privada. Esta traducción la realiza un proceso llamado NAT que se ejecuta en el router doméstico, y por eso se le llama «router NAT».

20.3.1. Traducción de Direcciones de Red (NAT)

NAT (Network Address Translation) [35] es un programa (software) que interconecta la red privada con la red del ISP, y traduce (reescribe) las direcciones de los paquetes IP que reenvía. La topología es similar a la Figura 20.4. El router NAT tiene una interfaz WAN (la que conecta con la red del ISP) con una dirección IP pública (*p. ej.* 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 nodos tienen direcciones privadas del mismo bloque 192.168.0.0/24 asignadas por el servidor DHCP.

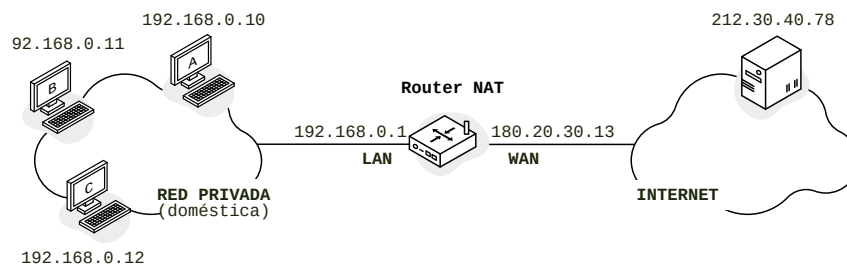


FIGURA 20.4: Ejemplo de configuración NAT en una red doméstica

El uso más habitual de NAT es el que permite a los nodos de la red privada establecer conexiones con servidores de la red pública. Es el llamado SNAT que se muestra en la Figura 20.5. Si por ejemplo, el nodo A quiere conectar con un servidor público, cuando sus paquetes IP salen de la red, el router substituye la dirección **origen**³ (privada) por la dirección pública (WAN) del router. Al volver la respuesta procedente del servidor remoto, el router substituye la dirección destino (que es la pública del router) de nuevo por la dirección privada del nodo (ver Figura 20.5). Ten en cuenta que este mecanismo es transparente para el servidor y para cualquier otro elemento en Internet, es decir, ni participan ni son conocedores de que esta traducción está ocurriendo.

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 soft-

³Por eso se llama *Source NAT*

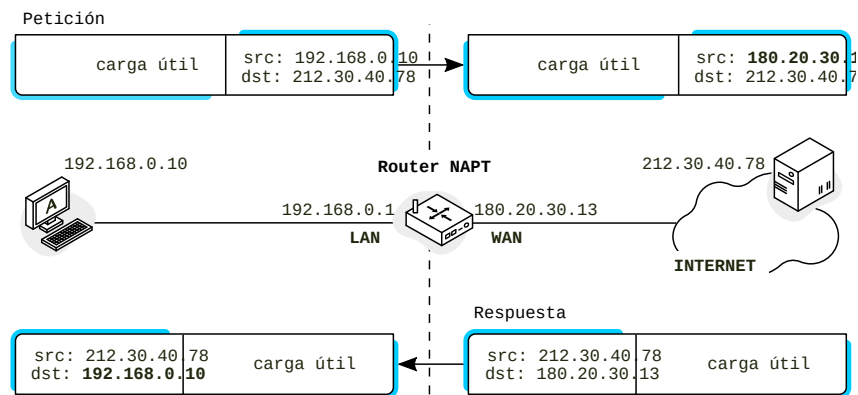


FIGURA 20.5: Ejemplo de SNAT

ware NAT apunta en la tabla NAT⁴ esas correspondencias en el momento de hacer la traducción de salida. Lógicamente, estas entradas en la tabla desaparecen tras un tiempo de inactividad o bien cuando se detecta el cierre de la conexión (si es que el flujo es TCP). El Cuadro 20.2 muestra la tabla tras dos conexiones desde los nodos A y B a dos servidores remotos diferentes.

Dirección local	Dirección remota
192.168.0.10	212.30.40.78
192.168.0.12	200.25.34.56

CUADRO 20.2: Tabla NAT para el envío de la Figura 20.5

Sin embargo, la información de esta tabla es insuficiente. Tiene una limitación muy importante. La traducción de direcciones en la respuesta es ambigua si dos o más nodos de la red privada contactan con el mismo servidor remoto a la vez. Es ambigua en el sentido de que el router no tiene forma de saber qué nodo privado fue el que envió la petición que corresponde a esta respuesta.

Para reducir esta ambigüedad existe una técnica mejorada denominada NAPT (Network Address Port Translation) [36]. Consiste en incorporar a la tabla NAT los puertos origen y destino TCP o UDP. La probabilidad de que dos nodos de la red privada elijan el mismo puerto origen en conexiones simultáneas a un mismo servidor remoto es realmente muy baja, y ese sería el único caso en el que se presentaría la ambigüedad. La Figura 20.6 y la Tabla 20.3 ilustran esta técnica.

⁴Formalmente: *Address Translation Table*

Dirección local	P. local	Dirección remota	P. remoto	Proto
192.168.0.10	4045	180.20.30.13	80	TCP
192.168.0.12	32400	130.0.23.12	22	TCP

CUADRO 20.3: Tabla NAPT para el envío de la Figura 20.6

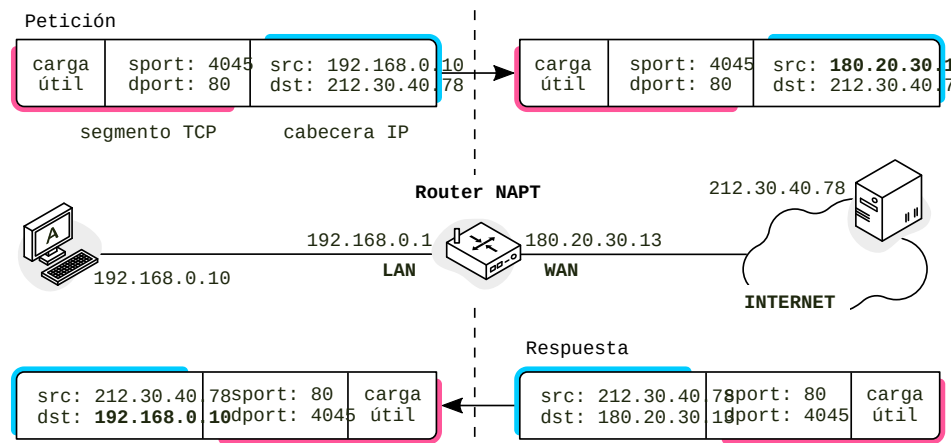


FIGURA 20.6: Ejemplo de NAPT

En todo caso, muchas implementaciones de NAT optan por evitar completamente la ambigüedad traduciendo el puerto del cliente local por un puerto ficticio que el router puede elegir garantizando así que será único en la tabla. Este puerto se conoce como «puerto mapeado» o «sintético». Esta situación se muestra en la Figura 20.7 y la Tabla 20.4.

Dirección local	P. local	P. mapeado	Dirección remota	P. remoto	Proto
192.168.0.10	4045	7312	180.20.30.13	80	TCP
192.168.0.12	32400	45012	130.0.23.12	22	TCP

CUADRO 20.4: Tabla NAPT con puertos mapeados para el envío de la Figura 20.6

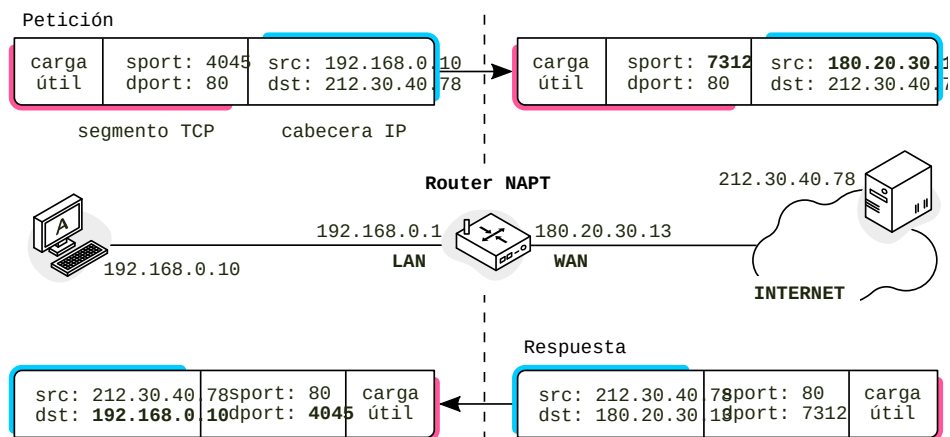
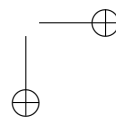
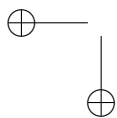
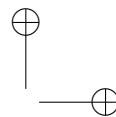
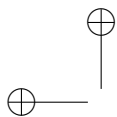


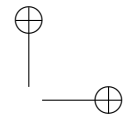
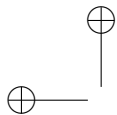
FIGURA 20.7: Ejemplo de NAPT con puertos mapeados

20.4. Reenvío de puertos

Otro inconveniente de NAT, que NAPT tampoco resuelve es que todo el mecanismo se basa en que el router aprende del tráfico saliente de la red privada, y usa esa información para realizar la traducción inversa. Eso implica que no es posible poner servidores en los nodos privados que puedan ser alcanzados desde clientes fuera de la red privada. Para resolver esto, el router proporciona un medio para incluir filas permanentes en la tabla NAT.

Estas entradas en la tabla se conocen como «reenvío de puertos» (*port forwarding*). Ten en cuenta que en este caso, el router traduce la dirección IP destino, en lugar de la origen, y por eso se denomina *Destination NAT* (DNAT). Como esta configuración está pensada para dar acceso a un servidor interno, lo habitual es que el usuario que crea una entrada DNAT en la tabla indique tanto el puerto destino del router como el puerto destino del nodo privado, que es donde realmente hay un servidor vinculado.





Capítulo 21

DNS

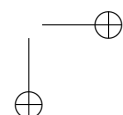
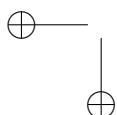
DNS es el servicio de nombres para redes TCP/IP, es decir, el sistema que asocia nombres únicos a direcciones IP, de forma similar a como una agenda asocia nombres a números de teléfono. Y esto es necesario —o muy conveniente— por varios motivos:

- Las direcciones IP son difíciles de recordar para las personas, especialmente las IPv6.
- Es sencillo asociar significados a los nombres, mientras que las direcciones no lo tienen. Eso facilita la identificación de los servicios que proporcionan determinados nodos (*p. ej.* `mail.example.com` probablemente aloja un servidor de correo).
- Tener nombres permite mantener referencias fijas para los nodos, permitiendo que las direcciones cambien. Eso permite al proveedor migrar el servidor (moverlo a otro nodo) sin que los usuarios, u otros servicios, lo noten¹.
- Un mismo nombre puede asociarse a varias direcciones, lo que proporciona redundancia, y con ello mejora de la disponibilidad, tolerancia a fallos y balanceo de carga.

Formalmente se denomina Sistema de Nombres de Dominio (Domain Name System). Pero ¿qué es un ‘dominio’? La definición oficial resulta un tanto... recursiva [37] y nada intuitiva. Por eso aquí lo vamos a enfocar de otro modo.

DNS define un espacio de nombres jerárquico organizado como un árbol invertido, es decir, con la raíz arriba. Cada nodo de este árbol tiene un nombre asociado. El dominio es el subárbol que se obtiene a partir de cualquiera de esos nodos. El *nombre de dominio* para un nodo se obtiene concatenando los nombres de los nodos que van desde ese hacia arriba hasta

¹Esto se denomina *transparencia de localización*.



la raíz, y por convenio esos nombres se separan con puntos. Por ejemplo, en la Figura 21.1, el dominio `ietf.org` está formado, aparte del propio nodo `ietf`, por los nodos `www`, `mail` y `dev`.

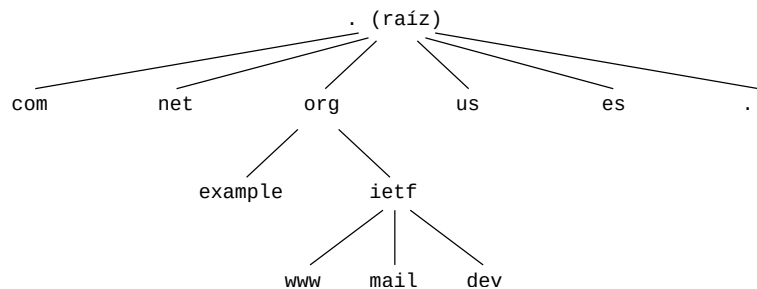


FIGURA 21.1: Estructura jerárquica de los nombres de dominio

Cualquier dominio puede tener subdominios si se toman subárboles a partir del nodo base que lo define. Es fácil entender porqué todo lo relacionado con DNS suena recursivo: el árbol es una estructura recursiva por definición.

Como puede ser confuso, aclaremos que DNS da nombre a varias cosas. Es el nombre del propio servicio distribuido que traduce nombres a direcciones, el del protocolo que define los mensajes que se intercambian los servidores y clientes, y también se utiliza para referirse a los servidores que implementan este servicio: Cuando se habla de «un DNS» se entiende que hablamos de un servidor concreto, que puede ser un proceso o un nodo, en función del contexto.

Conceptualmente, el servicio DNS funciona como una base de datos distribuida y redundante que almacena la información en forma de **registros** nombre-tipo-valor. Esto convierte el protocolo asociado en un mecanismo de consulta. A estas consultas las llamamos en español «resolución de nombres».

La resolución de nombres es una tarea muy frecuente, las aplicaciones lo hacen constantemente como parte de su funcionamiento. Lo hacen cada vez que demandan cualquier servicio de red. Pero también disponemos de herramientas para hacer esas consultas manualmente, y resultan muy útiles para diagnosticar problemas, probar entender la configuración o estudiar su funcionamiento. Una de estas herramientas es `dig`. Aquí puedes ver un ejemplo básico de uso:

```

$ dig www.ietf.org

; <<>> DiG 9.20.15-1-deb13u1-Debian <<>> www.ietf.org
;; global options: +cmd

```

```
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 12494
;; flags: qr rd ra ad; QUERY: 1, ANSWER: 2, AUTHORITY: 0, ADDITIONAL: 1

;; OPT PSEUDOSECTION:
; EDNS: version: 0, flags:; udp: 1232
;; QUESTION SECTION:
;ietf.org.                IN      A

;; ANSWER SECTION:
www.ietf.org.            113     IN      A      104.16.45.99
www.ietf.org.            113     IN      A      104.16.44.99

;; Query time: 7 msec
;; SERVER: 1.1.1.1#53(1.1.1.1) (UDP)
;; WHEN: Tue Jan 06 14:33:58 UTC 2026
;; MSG SIZE rcvd: 69
```

Sin embargo `dig` produce mucha información que puede no ser relevante. Por suerte, acepta varias opciones para limitar su salida. Además puedes escribir algunas de esas opciones en el archivo `~/.digrc` para que el programa las aplique siempre y así no tener que escribirlas cada vez. Por ejemplo, para este capítulo usamos un archivo `~/.digrc` que contiene lo siguiente:

```
+nocmd
+noedns
+noquestion
+nostats
```

Con eso, si repites el comando anterior, la salida será algo así:

```
$ dig www.ietf.org
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 6684
;; flags: qr rd ra ad; QUERY: 1, ANSWER: 2, AUTHORITY: 0, ADDITIONAL: 0

;; ANSWER SECTION:
www.ietf.org.            113     IN      A      104.16.45.99
www.ietf.org.            113     IN      A      104.16.44.99
```

Ahora que la salida es manejable, analicemos su contenido. Este comando en concreto pregunta al servidor DNS configurado en tu sistema acerca del dominio `www.ietf.org`. La respuesta (`ANSWER SECTION`) informa de dos registros tipo A. Estos indican las direcciones IPv4 asociadas: `104.16.44.99` y `104.16.45.99`. Quizá te sorprenda que sean dos y no una, pero eso es perfectamente posible y, de hecho, bastante común. Eso mejora la disponibilidad del servicio, porque el cliente normalmente usará la primera, pero si falla, empleará la segunda.

Además, el servidor puede reordenar esos registros cada vez que se le piden, mejorando el balanceo de la carga entre ambos nodos. Repite el comando varias veces y fíjate en el orden en que aparecen las direcciones cambia en

cada ocasión. Más adelante veremos cómo se saca provecho de esta característica.

Ya que estamos, veamos qué significa la información que aparece en la cabecera:

- opcode: QUERY: Se trata de una consulta estándar.
- status: NOERROR: La consulta se ha ejecutado sin errores.
- flags: qr rd ra ad:
 - qr: El mensaje es una respuesta (*query response*).
 - rd: El cliente ha solicitado resolución recursiva (*recursion desired*).
 - ra: El servidor soporta resolución recursiva (*recursion available*).
 - ad: Los datos están autenticados (*authenticated data*).

Antes hemos hablado del «servidor DNS configurado en tu sistema». Hablamos de la dirección de un servidor —en realidad suelen ser dos— que el SO utiliza para realizar las consultas DNS. Normalmente se obtienen automáticamente con DHCP, así que no es algo que nos suela preocupar. En § 21.3 veremos cómo se pueden fijar de forma manual, y por qué puede ser interesante.

21.1. Dominios y zonas

El nombre que acabamos de utilizar —`www.ietf.org`— es efectivamente un dominio, pero también es un subdominio de `ietf.org`, que a su vez es un subdominio de `net`. El nombre `net` es lo que se llama un dominio de nivel superior o TLD (Top Level Domain). Los TLD cuelgan directamente de la raíz del árbol, que se denota con un punto (.) al final del nombre, como en `www.ietf.org.` —aunque ese punto final normalmente se omite en los usos menos formales. Un nombre de dominio de este tipo (que termina en el raíz) se denomina FQDN (Fully Qualified Domain Name), que se suele traducir como *nombre de dominio completamente calificado*.

Esta nomenclatura es meramente conceptual y no tiene relación con el modo en que se almacena la información en los servidores. Para cada dominio existe al menos un servidor específico que define la correspondencia oficial entre nombres y direcciones IP para ese dominio —si bien suelen ser dos por redundancia. De estos servidores se dice que son los **autoritativos** del dominio. Como ejemplo, veamos cuáles son los servidores autoritativos para el dominio `ietf.org`, pidiendo los registros NS.

```
$ dig ietf.org NS
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 45975
;; flags: qr rd ra ad; QUERY: 1, ANSWER: 2, AUTHORITY: 0, ADDITIONAL: 0
```

```
;; ANSWER SECTION:
ietf.org.      86400    IN       NS       ken.ns.cloudflare.com.
ietf.org.      86400    IN       NS       jill.ns.cloudflare.com.
```

Miles de servidores (no autoritativos) repartidos por todo el mundo pueden resolver también el nombre `ietf.org`, pero eso es simplemente porque tienen una copia de la información que proporcionan estos servidores autoritativos.

El servidor autoritativo define la llamada *zona* del dominio, que incluye los registros asociados, y responde a consultas específicas sobre esta información. La zona puede incluir la definición de los subdominios o bien, puede delegarlos a servidores autoritativos específicos. Sin embargo, los servidores autoritativos no proporcionan información sobre dominios ajenos, únicamente sobre el que administran.

Veamos los tipos de registro esenciales que se pueden definir en una zona DNS.

Tipo	Significado	Descripción
SOA	Start of Authority	Indica el servidor autoritativo primario (<i>master</i>), correo del administrador, número de serie, etc.
NS	Name Server	Indica los servidores autoritativos de la zona o delega sus subdominios.
A	Address	Asocia un nombre con una dirección IPv4.
CNAME	Canonical Name	Define un alias para otro nombre.

CUADRO 21.1: Registros básicos para definir zonas DNS

Aparte de los NS, que ya hemos visto, el registro más importante es el SOA, que define parámetros esenciales de la zona. Veamos su aspecto real. La Tabla 21.2 detalla el significado de los campos incluidos en el registro.

```
$ dig +multiline ietf.org SOA
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 49304
;; flags: qr rd ra ad; QUERY: 1, ANSWER: 1, AUTHORITY: 0, ADDITIONAL: 0

;; ANSWER SECTION:
ietf.org.      1800 IN    SOA jill.ns.cloudflare.com. dns.cloudflare.com. (
                2392675761 ; serial
                10000    ; refresh (2 hours 46 minutes 40 seconds)
                2400    ; retry (40 minutes)
                604800   ; expire (1 week)
                1800    ; negative TTL (30 minutes)
                )
```

Campo	Significado
Primary NS	Servidor autoritativo primario declarado para la zona: <code>jill.ns.cloudflare.com</code>
Responsible Mail	Correo del administrador, aunque no siempre tiene formato de correo electrónico.
Serial	Número de serie de la zona.
Refresh (s)	Cada cuanto un secundario comprueba si hay cambios.
Retry (s)	Tiempo de reintento si falla el refresco.
Expire (s)	Tras este tiempo sin contacto, el secundario deja de servir la zona.
Negative TTL (s)	Tiempo de caché para respuestas negativas.

CUADRO 21.2: Valores reales e interpretación del registro SOA de `ietf.org`

21.2. La zona raíz

Como para cualquier otro dominio, el dominio raíz también tiene su propia zona. La zona raíz gestiona los registros NS para todos los TLD. Está coordinada por la ICANN y la IANA, y consta de 13 servidores replicados a centenares por todo el mundo. Están nombrados desde `a.root-servers.net` a `m.root-servers.net`.

Puedes consultar la zona raíz y ver los nombres de esos servidores, y también su SOA:

```
$ dig . NS
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 23022
;; flags: qr rd ra ad; QUERY: 1, ANSWER: 13, AUTHORITY: 0, ADDITIONAL: 0

;; ANSWER SECTION:
.          510508    IN      NS      a.root-servers.net.
.          510508    IN      NS      b.root-servers.net.
.          510508    IN      NS      c.root-servers.net.
.          510508    IN      NS      d.root-servers.net.
.          510508    IN      NS      e.root-servers.net.
.          510508    IN      NS      f.root-servers.net.
.          510508    IN      NS      g.root-servers.net.
.          510508    IN      NS      h.root-servers.net.
.          510508    IN      NS      i.root-servers.net.
.          510508    IN      NS      j.root-servers.net.
.          510508    IN      NS      k.root-servers.net.
.          510508    IN      NS      l.root-servers.net.
.          510508    IN      NS      m.root-servers.net.

$ dig . SOA
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 45090
;; flags: qr rd ra ad; QUERY: 1, ANSWER: 1, AUTHORITY: 0, ADDITIONAL: 0

;; ANSWER SECTION:
```

```
.      86351      IN      SOA      a.root-servers.net. nstld.verisign-grs.com.
2026010600 1800 900 604800 86400
```

21.3. Resolución de nombres

En los nodos de los usuarios, al componente que realiza las resoluciones de nombres lo llamaremos *resolvedor* como calco léxico del original en inglés *local resolver* o *stub resolver*. Suele ser de una librería o servicio del SO que recibe y responde peticiones de parte de las aplicaciones a través de la llamada al sistema `getaddrinfo()` o de las librerías que cada lenguaje de programación ofrezca por encima. Por ejemplo, en Python:

```
import socket
for addr in socket.getaddrinfo("www.example.com", None):
    print(addr[4][0])
104.18.5.106
104.18.4.106
```

El módulo `socket` también utiliza automáticamente el resolvedor del SO cuando en lugar de una IP, utilizas un nombre al establecer una conexión TCP o flujo UDP, como en:

```
import socket
s = socket.socket()
s.connect(("www.example.com", 80))
```

El resolvedor determina las fuentes de información que debe usar consultando la línea `hosts` en el archivo `/etc/nsswitch.conf`, que suele tener este aspecto:

```
hosts: files dns
```

Ese «files» indica que primero debe consultar el archivo `/etc/hosts`, que contendrá algo similar a esto:

```
127.0.0.1    localhost
127.0.0.1    maine
::1         ip6-localhost ip6-loopback
ff02::1     ip6-allnodes
ff02::2     ip6-allrouters
```

Si no encuentra el nombre aquí, la palabra «dns» de `nsswitch.conf` indica que debe consultar los servidores DNS configurados en el archivo `resolv.conf`, que suele tener este aspecto:

```
nameserver 1.1.1.1
nameserver 8.8.8.8
```

Estas direcciones IP corresponden a los servidores DNS llamados *recursive resolvers*, y que aquí llamaremos simplemente «servidores DNS» para distinguirlos de los autoritativos. Estos servidores son los encargados de realizar las consultas necesarias para resolver el nombre solicitado por el cliente y suelen proporcionarlos los ISP. Por eso, el contenido de este fichero normalmente se obtiene por DHCP u otros servicios locales como `systemd-resolved` o `resolvconf`.

También existe la posibilidad de editarlo manualmente si ninguno de esos servicios está en uso, o si se quiere forzar un servidor específico ajeno al ISP, como es el caso del ejemplo. La Figura 21.2 muestra cómo hacer dicha configuración en el entorno de escritorio GNOME.

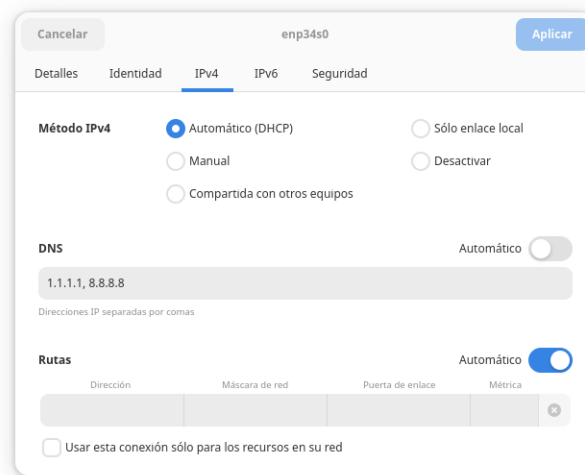


FIGURA 21.2: Configuración DNS en GNOME

La razón por la que se establecen dos de estos servidores es la habitual: redundancia. Si el primero no responde u ocasionalmente lo hace muy lento, el SO puede utilizar el segundo.

Normalmente el cliente pide a estos servidores una *resolución recursiva*, que implica los siguientes pasos del servidor DNS:

1. Si el nombre solicitado está en su caché, responde inmediatamente al cliente.
2. Si no está en la caché, realiza una consulta a un servidor raíz, que le indica el servidor autoritativo para el TLD solicitado.

3. Pregunta al servidor autoritativo del TLD, que a su vez delega la consulta al servidor autoritativo del dominio solicitado.
4. Realiza una consulta al servidor autoritativo específico, que puede disponer de la dirección o bien responder con otra delegación: el registro NS de un subdominio.
5. Guarda la respuesta en caché y la devuelve al cliente.

Es *recursivo* porque el servidor (y no el cliente) se encarga de seguir las delegaciones, es decir, realiza las consultas necesarias hasta llegar hasta la respuesta final. Por defecto, el cliente solicita resoluciones recursivas al servidor (RD=1 en la cabecera del mensaje DNS), pero opcionalmente es posible hacer solicitudes iterativas (RD=0). En ese caso, el servidor responde con la información de que dispone, pero no hace peticiones adicionales. Lo importante es entender que el funcionamiento típico del servidor DNS es ofrecer resolución recursiva realizando peticiones iterativas.

Para entender mejor el trabajo que realiza el servidor, podemos ver un proceso similar forzando una resolución iterativa desde el cliente:

```

1  $ dig +trace www.ietf.org
2  .          513823   IN      NS      a.root-servers.net.
3  .          513823   IN      NS      b.root-servers.net.
4  .          513823   IN      NS      c.root-servers.net.
5  .          513823   IN      NS      d.root-servers.net.
6  .          513823   IN      NS      e.root-servers.net.
7  .          513823   IN      NS      f.root-servers.net.
8  .          513823   IN      NS      g.root-servers.net.
9  .          513823   IN      NS      h.root-servers.net.
10 .          513823   IN      NS      i.root-servers.net.
11 .          513823   IN      NS      j.root-servers.net.
12 .          513823   IN      NS      k.root-servers.net.
13 .          513823   IN      NS      l.root-servers.net.
14 .          513823   IN      NS      m.root-servers.net.
15 ;; Received 525 bytes from 1.1.1.1#53(1.1.1.1) in 7 ms
16
17 org.        172800   IN      NS      a0.org.afiliast-nst.info.
18 org.        172800   IN      NS      a2.org.afiliast-nst.info.
19 org.        172800   IN      NS      b0.org.afiliast-nst.org.
20 org.        172800   IN      NS      b2.org.afiliast-nst.org.
21 org.        172800   IN      NS      c0.org.afiliast-nst.info.
22 org.        172800   IN      NS      d0.org.afiliast-nst.org.
23 ;; Received 812 bytes from 2001:7fe::53#53(i.root-servers.net) in 191 ms
24
25 ietf.org.    3600     IN      NS      ken.ns.cloudflare.com.
26 ietf.org.    3600     IN      NS      jill.ns.cloudflare.com.
27 ;; Received 306 bytes from 199.19.54.1#53(b0.org.afiliast-nst.org) in 19 ms
28
29 www.ietf.org. 300      IN      A       104.16.45.99
30 www.ietf.org. 300      IN      A       104.16.44.99
31 ;; Received 177 bytes from 173.245.58.122#53(jill.ns.cloudflare.com) in 3 ms

```

Este comando se ha ejecutado teniendo configurado 1.1.1.1 en el archivo `/etc/resolv.conf`. Veamos las diferentes partes que la componen, que son las respuestas a las consultas sucesivas que hace el cliente a partir de la información que va recibiendo de cada servidor:

- Consulta al servidor raíz (**línea 2**). Responde con los 13 servidores autoritativos para el dominio raíz.
- Consulta a `i.root-servers.net` acerca del TLD `org`, que responde con 6 servidores autoritativos (**línea 17**).
- Consulta a `b0.org.afiliat-nst.org` sobre `ietf.org` (**línea 25**). Responde con dos servidores autoritativos.
- Consulta a `jill.ns.cloudflare.com` para resolver `www.ietf.org`, que finalmente responde con las direcciones IP solicitadas (**línea 29**).

Después de cada sección aparece el servidor concreto al que se ha consultado y cuanto tiempo tardó en responder.

21.4. Cachés e información obsoleta

Las cachés de los servidores DNS juegan un papel clave en el funcionamiento del sistema, proporcionando un equilibrio entre disponibilidad de información actualizada y rendimiento. Cuando un servidor recibe una petición desde un cliente, primero comprueba su caché. Si tiene la respuesta y aún es válida (no ha expirado), devuelve inmediatamente ese valor y no realiza consultas adicionales. Sin la caché, los servidores continuamente tendrían que hacer consultas a los mismos servidores autoritativos, lo que conllevaría una sobrecarga inabordable para todas las partes y degradaría de forma muy notable la operación del sistema en su conjunto.

La invalidación de la caché la determina el campo TTL (expresado en segundos) asociado a cada registro. El servidor autoritativo define el valor del TTL por lo que no cambia, a menos que el administrador de la zona lo modifique manualmente. Puedes averiguar el valor del TTL tal como está definido dirigiendo la consulta directamente al servidor autoritativo (usando la `@`) obviando el servidor configurado en tu sistema.

```
$ dig @jill.ns.cloudflare.com ietf.org
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 11996
;; flags: qr aa rd; QUERY: 1, ANSWER: 2, AUTHORITY: 0, ADDITIONAL: 0
;; WARNING: recursion requested but not available

;; ANSWER SECTION:
ietf.org.      300    IN     A      104.16.45.99
ietf.org.      300    IN     A      104.16.44.99
```

Hay dos pistas en esta respuesta que confirman que hemos preguntado al servidor autoritativo:

- En la cabecera aparece el flag aa (*authoritative answer*).
- La advertencia indicando *recursion requested but not available*, algo que ya hemos explicado es lo esperable para servidores autoritativos.

El TTL es el valor que aparece en la segunda columna junto a cada registro (300 segundos en este caso). Si repites esta misma consulta verás que siempre aparece el mismo valor, mientras que si envías la consulta al servidor proporcionado por el ISP, el valor irá disminuyendo hasta llegar a cero, momento en el que hará una nueva consulta al autoritativo y volverá a empezar desde 300. Este sistema tiene una consecuencia importante: la información que ofrecen los servidores siempre está potencialmente obsoleta². ¿Cuánto? Depende completamente del valor del TTL definido en cada zona, si bien es cierto que muchos ISP imponen un valor mínimo (5 minutos) y un valor máximo (1 día).

Y así es cómo se distribuye la información a través de la red DNS. No hay ningún mecanismo que detecte cambios y sea proactivo en la distribución de actualizaciones. Cuando por ejemplo el administrador de una zona modifica la dirección IP asociada a un nombre, los servidores seguirán ofreciendo información incorrecta a sus clientes hasta que sus cachés expiren.

Es común escuchar que las asociaciones DNS «tardan en propagarse», pero en realidad solo depende de la configuración del TTL del servidor autoritativo y de los servidores de terceros que estén involucrados. Por ese motivo, cuando se realiza una migración que implica un cambio de dirección IP es buena práctica reducir el TTL horas o días antes, dando tiempo para que expiren las cachés en toda la red. Una vez terminada la migración, puede volver a aumentarse el TTL para evitar consultas demasiado frecuentes.

La información negativa, es decir, la situación en la que un servidor autoritativo dice que no conoce un nombre, también se almacena en caché. En ese caso se utiliza el valor llamado *Negative TTL* que aparece en el registro SOA (ver Cuadro 21.2). Así se evitan consultas reiteradas para nombres desconocidos a los mismos servidores.

²Se dice que ofrece *consistencia eventual*

21.5. Configuración de zona

La configuración de una zona en un servidor autoritativo se especifica en un archivo llamado *fichero de zona*³ que contiene los registros que la definen.

El siguiente listado muestra una definición de zona hipotética para el dominio `example.net`⁴. Este fichero de zona se aplicaría en los servidores autoritativos y para responder a las consultas sobre ese dominio.

```
$TTL 3600
@ IN SOA ns1.example.net. ns2.example.net. (
    2026012901 ; serial
    3600      ; refresh
    900       ; retry
    604800    ; expire
    300       ; negative TTL
)

@ IN NS      ns1.example.net.
@ IN NS      ns2.example.net.

@ IN A       192.0.2.10
@ IN AAAA    2001:db8::10

www IN CNAME @
@ IN MX 10 mail.example.net.
_sip._tcp IN SRV 10 60 5060 sip.example.net.
```

Fíjate que ambos servidores autoritativos sirven la misma información, ambos definen el registro SOA que apunta al servidor primario (`ns1`). Ambos incluyen los dos registros NS.

Esta zona define también:

- El valor por defecto de TTL para los registros que no lo especifiquen explícitamente (3600 segundos).
- Registro A para el dominio base⁵ `example.net` (representado por '@').
- Registro AAAA para el dominio base, que proporciona una dirección IPv6 asociada.
- Registro CNAME que define un alias `www.example.net` para el dominio base, es decir, devuelve la misma dirección IP.
- El registro MX que define el servidor de correo para el dominio.
- El registro SRV que define el servicio SIP sobre TCP para el dominio.

³O alguna estructura de datos equivalente.

⁴El dominio `example.net` existe realmente con otra configuración

⁵También se denomina en inglés *apex domain* o *root domain*, pero evitaremos esta segunda opción porque se puede confundir con el dominio raíz: '.'

Para que esta zona sea legítima y se pueda utilizar, la zona del `net.` debe definir obligatoriamente los mismos valores para los registros NS. Los registros correspondientes en el fichero de zona de `net.` serían algo así:

```
example.net.      3600  IN  NS   ns1.example.net.
example.net.      3600  IN  NS   ns2.example.net.
ns1.example.net.  IN  A    192.0.2.53
ns2.example.net.  IN  A    192.0.2.54
```

Los registro NS representan aquí la delegación del dominio `example.net` a los servidores autoritativos indicados (`ns1` y `ns2`). Los registros A adicionales se denominan *registros glue* y son necesarios para evitar un posible bucle de resolución cuando el nombre del servidor autoritativo (como en este caso) está dentro del dominio delegado.

21.6. Resolución inversa

ToDo!

21.7. Replicación

Algunas compañías ofrecen servidores DNS públicos que proporcionan algunas ventajas sobre los que facilitan los ISP. Esta tabla recoge los más conocidos, sus direcciones IP, compañía y propósito principal:

IP	Compañía	Propósito
8.8.8.8 / 8.8.4.4	Google	Rapidez y estabilidad.
1.1.1.1 / 1.0.0.1	Cloudflare	Privacidad y baja latencia.
9.9.9.9	Quad9 Foundation	Bloqueo contra dominios phishing.
208.67.222.123 / 208.67.220.123	Cisco (OpenDNS FamilyShield)	Control parental.
208.67.222.222 / 208.67.220.220	Cisco (OpenDNS Home)	Filtrado configurable y es...
185.228.168.9 / 185.228.169.9	CleanBrowsing	Bloqueo de malware.
185.228.168.168 / 185.228.169.168	CleanBrowsing	Control parental.
94.140.14.14 / 94.140.15.15	AdGuard	Bloqueo de publicidad.
84.200.69.80 / 84.200.70.40	DNS.Watch	Neutralidad y privacidad.
156.154.70.1 / 156.154.71.1	Neustar	Protección contra dominio...
64.6.64.6 / 64.6.65.6	Verisign	Estabilidad y privacidad.

Cosas como la protección contra malware, phishing o control parental la consiguen simplemente manteniendo una lista negra de dominios y respondiendo con un registro A falso o un dominio que lleva a una página que informa del bloqueo. Pero ¿cómo puede un servidor DNS ofrecer baja latencia a usuarios de todo el planeta? Por lógica, el servidor ofrecerá menor

latencia a los usuarios que estén próximos geográficamente que a aquellos que estén lejos. La respuesta es que en realidad el servidor DNS no está en un nodo, sino que hay decenas o centenares de servidores (réplicas) con la misma dirección IP repartidos por el mundo en instalaciones PoP (Point of Presence) bajo el control administrativo de la compañía. De este modo, la latencia es baja para cualquier usuario, en cualquier lugar.

Pero eso nos lleva a otra cuestión aún más básica: ¿cómo puede haber centenares de nodos con la misma dirección IP? Esto es posible con *anycast*, una técnica de encaminamiento que puede llevar paquetes IP al nodo más próximo (mejor métrica) con la dirección destino solicitada. Realmente IPv4 no soporta *anycast* como sí lo hace IPv6, pero en la práctica el encaminamiento BGP puede lograr el mismo efecto.

21.7.1. CDN

Una CDN es un servicio de caché distribuida para recursos web estáticos: archivos JavaScript, CSS, imágenes, etc. El objetivo, como de costumbre, es aumentar la disponibilidad y reducir la latencia para el acceso desde los usuarios finales a estos recursos.

Para lograr esto, cuando un usuario pide la resolución del dominio de una CDN (*p. ej.* `ajax.googleapis.com`), el servidor autoritativo (gestionado por la CDN) responde con una dirección *anycast* o bien con la dirección IP de un PoP próximo al usuario, en cuyo caso además, puede tener en cuenta la carga a la hora de elegirlo, que es algo que no puede hacer *anycast* únicamente mediante encaminamiento BGP.

En este ejemplo se puede apreciar cómo, al caducar la caché, el servidor DNS configurado en el sistema empieza a devolver una dirección diferente.

```
~$ dig ajax.googleapis.com
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 17254
;; flags: qr rd ra; QUERY: 1, ANSWER: 1, AUTHORITY: 0, ADDITIONAL: 1

;; ANSWER SECTION:
ajax.googleapis.com.      1      IN      A       142.250.200.106

\~$ dig ajax.googleapis.com
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 13269
;; flags: qr rd ra; QUERY: 1, ANSWER: 1, AUTHORITY: 0, ADDITIONAL: 1

;; ANSWER SECTION:
ajax.googleapis.com.     290    IN      A       216.58.209.74
```

El sistema CDN también utiliza un mecanismo de caché con expiración, aunque en este caso se aprovechan los campos específicos de las cabeceras HTTP para determinar la validez de los recursos en caché. Cuando el objeto

solicitado no se encuentra o ha caducado, el nodo CDN *edge* lo obtiene del servidor origen o de un nodo intermedio de la red CDN. Los nodos intermedios se utilizan en redes CDN jerárquicas, algo habitual cuando esta alcanza un tamaño importante.

21.8. Transporte

El protocolo DNS original se encapsula sobre UDP y el servidor escucha en el puerto 53. Este sigue siendo el transporte más usado también hoy día, especialmente para consultas desde el cliente. El motivo es que UDP es ligero y de baja latencia, algo muy importante para un servicio como este que se utiliza a menudo en sesiones muy cortas.

Sin embargo, existen otros transportes estandarizados:

- TCP en el puerto 53, utilizado para transferencia de zona entre servidores autoritativos y para consultas cuyas respuestas que no caben en un único datagrama UDP.
- DNS sobre TLS (DoT) en el puerto 853, que proporciona privacidad y cifrando en consultas y respuestas.
- DNS sobre HTTPS (DoH) en el puerto 443, que también proporciona privacidad y cifrado, pero además permite *disimular* las consultas DNS dentro de tráfico HTTPS incluso sobre HTTP/2 o HTTP/3.
- DNS sobre QUIC (DoQ) en el puerto 853. Utiliza QUIC que es fiable, cifrado y con control de congestión sobre UDP.

21.9. Multicast DNS

Multicast DNS, abreviado como mDNS, permite la resolución de nombres en redes LAN sin necesidad de servidores. Utiliza el mismo formato de mensajes DNS estándar, pero los clientes envían las solicitudes a la dirección multicast 224.0.0.251, puerto 5353.

Los nodos se autoasignan nombres basados en su hostname en el dominio especial `.local`. Este mecanismo es la base de *zero-configuration networking* (*zeroconf*) que se utiliza en redes LAN para la configuración y, sobre todo, descubrimiento automático de servicios y dispositivos gracias a DNS-SD.

Con DNS-SD y mDNS es posible anunciar y descubrir impresoras, servidores de ficheros, televisores, altavoces, dispositivos IoT, etc. Se utilizan nombres de servicio con el formato `_servicio._protocolo.local`, por ejemplo `_ssh._tcp.local` para un servidor SSH o `_ipp._tcp.local` para una impresora. La información se proporciona con los registros estándar PTR, SRV y TXT.

En GNU/Linux ambos protocolos se implementan con el sistema `avahi`. Por ejemplo, el siguiente comando lista los servicios que se están anunciando en la LAN:

```
$ avahi-browse -a
+ wlp1s0 IPv4 shellyplus1-c83d56432      Web Site      local
+ wlp1s0 IPv4 TV                        AirPlay Remote Video local
+ wlp1s0 IPv4 Google-Cast-Group-c262345234523 _googlecast._tcp local
```

Y se puede obtener información detallada:

```
= wlp1s0 IPv4 TV                        AirPlay Remote Video local
  hostname = [tv.local]
  address = [192.168.0.173]
  port = [51620]
  txt = ["serialNumber=0CAK3SDMA87463F" "manufacturer=Samsung" "model=QRQ60"
        "flags=0x244" "fv=p20.0.1" "rsf=0x3" "features=0x7F8AD0,0x38BCB46"
        "deviceId=D5:7D:D0:DF:D5:46" "acl=0"]
```

21.10. Dynamic DNS

Un servicio de DDNS lo proporciona un servidor DNS autoritativo que ofrece al administrador la posibilidad de modificar dinámicamente los registros de la zona (específicamente los A) normalmente a través de un API. Un servicio que se ejecuta en el nodo detecta automáticamente cuando cambia su dirección IP y lo notifica inmediatamente al servidor DDNS mediante ese API para que actualice la resolución.

Esta funcionalidad obviamente es útil para nodos con direcciones IP dinámicas, como típicamente proporcionan los ISP a sus clientes domésticos o pequeñas oficinas. Esto permite disponer de un nombre de dominio público y fijo que apunta a un nodo que no dispone de una dirección IP fija.

Existen varios servicios comerciales que ofrecen este servicio, tales como `no-ip.com`, `dyn.com`, `duckdns.org` o `freedns.afraid.org`. Aunque existe un protocolo estándar para este propósito [38], la mayoría usan API propietarias basadas en REST HTTP.

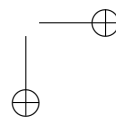
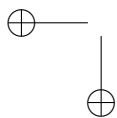
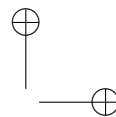
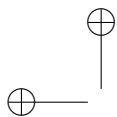
21.11. Servidores DNS *sinkhole*

Un sumidero DNS (*sinkhole*) es un servidor muy sencillo que se suele instalar en algún nodo de la LAN. Al igual que algunos de los servidores de la sección anterior, su objetivo es bloquear el acceso a dominios maliciosos o conocidos por participar en campañas de *phishing*, *spam*, publicidad, etc. La diferencia principal es que al ejecutarse en nuestra LAN tenemos plena capacidad para administrar con precisión el tipo de bloqueo que realizan. Algunos de los más conocidos son Pi-hole y AdGuard Home, que además están disponibles como contenedores docker con lo que su instalación resulta muy sencilla.



Los nombres de dominio más cortos del mundo

2.am n.nu b.bb c.cc g.gg t.tt b.cg b.mw e.mu m.tv n.tv r.tl 1.cm 2.cl
5.kg 7.ms p.ro e.tc s.ki o.ma v.pn c.sh b.mp



Capítulo 22

SSH

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 archivos², redireccionar conexiones, crear túneles que pueden transportar datos de conexiones arbitrarias o simplemente como sistema de autenticación.

22.1. 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.

Para probar SSH vamos a utilizar un contenedor docker. Para construir la imagen docker correspondiente, descarga el repositorio `git` con el siguiente comando:

```
alice@maine:~$ git clone https://github.com/destripando-internet/code
```

Y ejecuta el siguiente comando dentro del directorio `ssh-docker`. Esto crea la imagen y arranca el contenedor que ejecuta el servidor SSH:

¹...e inseguro

²con `scp`

```
alice@maine:~$ ~/net-book-code/ssh-docker$ docker-compose up -d
[...]
```

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).

```
alice@maine:~$ ssh user@172.20.0.2
user@172.20.0.2's password:
user@viper:~$
```

El comando `ssh` trata de establecer una conexión segura con el servidor SSH que hay en 172.20.0.2 (el contenedor docker). Concretamente queremos acceder con la cuenta del usuario `user`. Este proceso por defecto pide la contraseña del sistema asociada a ese usuario, que no se muestra al introducirla. En nuestro caso es **secret**.

22.2. Configuración

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

```
Host viper
  hostname 172.20.0.2
  User user
```

Es importante señalar que este archivo 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:

```
alice@maine:~$ ssh viper
user@172.20.0.2's password:
```

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

22.3. 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 coloca en el sistema al que quieres acceder y, de hecho, no hay ningún problema en que cualquiera la pueda leer o copiar (por eso es pública). Sin embargo, la clave privada debe quedar custodiada por el usuario y nunca debe ser accesible para nadie más.

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

```
alice@maine:~$ 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/EPiXEBfgqK3HLSqvphSeh60+qm9tz1DM alice@maine
```

Esto genera dos archivos 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:

```
alice@maine:~$ ssh-copy-id viper
user@172.20.0.2's password:

Number of key(s) added: 1

Now try logging into the machine, with:  "ssh 'viper'"
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 archivo `~/.ssh/authorized_keys` del servidor (viper en este ejemplo).

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

```
alice@maine:~$ ssh viper
Last login: Mon Mar 14 21:07:15 2022 from 172.20.0.1
user@viper:~$
```

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 en el servidor y ver la salida directamente en el computador cliente. Simplemente hay que escribir el comando a continuación:

```
alice@maine:~$ ssh viper ls /home/
user
```

22.4. Autenticación con certificado

Otra forma de autenticar un usuario es mediante 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 ca_key -P "" -C ca_key
Generating public/private rsa key pair.
Your identification has been saved in ca_key
Your public key has been saved in ca_key.pub
The key fingerprint is:
SHA256:nbN+pdYB3paUUnoPe60KJbkC3f1KiXCVbSIH51wQXbo ca_key
```

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

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

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

22.4.1. 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 `user` de la máquina `viper` como lo haría su administrador. Primero creamos un par de claves pública/privada para el usuario, con el nombre `user\_key`.

```
$ ssh-keygen -f user_key
```

Después, se firma la clave pública del usuario (`user\_key.pub`) con la clave privada de la CA (`ca\_key`):

```
$ ssh-keygen -s ca_key -I viper -n user user_key.pub
```

Con este comando se ha generado también el certificado `user\_key-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 *user*).

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

```
root@viper:/etc/ssh# chmod 644 ca_key.pub
root@viper:/etc/ssh# ls -la ca_key.pub
-rw-r--r-- 1 user user 562 Mar 3 09:13 ca_key.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/ca_key.pub
```

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

```
root@viper:/home/user# systemctl restart sshd
```

Con esto termina la configuración el servidor (*viper* en este ejemplo). Este proceso lo realizar automáticamente la construcción de la imagen docker y se puede ver en el archivo `Dockerfile`.

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

```
alice@maine:~$ ls -l ~/.ssh
-rw----- 1 ana ana 2590 mar 3 09:35 user_key
-rw-rw-r-- 1 ana ana 2020 mar 3 09:40 user_key-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 *viper*:

```
alice@maine:~$ ssh -i ~/.ssh/user_key viper
```

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

```
Host viper
  Hostname localhost
  Port 2200
  User user
  IdentityFile ~/.ssh/user_key
```

Y con esto se podrá acceder simplemente con:

```
alice@maine:~$ ssh viper
```

La diferencia respecto a lo explicado en la sección 22.3 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 ³, de modo que no hay posibilidad de usar `ssh-copy-id`. Esto se considera más seguro.

22.5. Copia de archivos con SCP

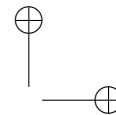
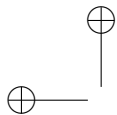
SCP (Secure Copy Protocol) es un protocolo de copia segura de archivos entre computadores cualesquiera conectados a Internet. Resulta especialmente cómodo ya que el propio servidor de `OpenSSH` lo incorpora y está activado por defecto.

Con la configuración previa es tan sencillo como:

```
$ scp un-\file viper:
un-\file                               100%   5   16.6KB/s   00:00
```

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

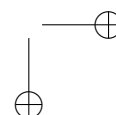
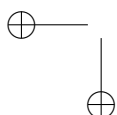
³PasswordAuthentication no

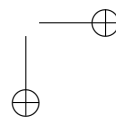
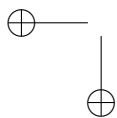
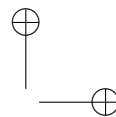
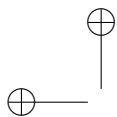


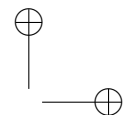
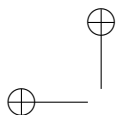
Capítulo 23

Epílogo

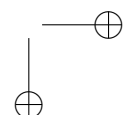
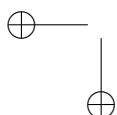
Este libro es el resultado de nuestra experiencia como docentes en las asignaturas de redes de computadores y sistemas distribuidos en la Escuela Superior de Informática de la UCLM. Como tal, esta experiencia se amplía y cambia cada curso, probando nuevas ideas y enfoques, por lo que el libro a buen seguro también lo hará, corrigiendo erratas y errores, actualizando y ampliando contenido, ejemplos y referencias.

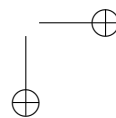
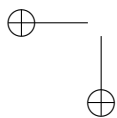
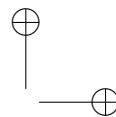
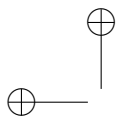






ANEXOS





Anexo A

Comandos habituales

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

A.1. Ficheros y directorios

cp (*copy*)

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

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

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

diff muestra las diferencias entre dos archivos.

find encuentra archivos a partir de su nombre.

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

```
alice@maine:~$ 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 archivos o directorios.

md5sum

calcula (o comprueba) la suma MD5 del archivo indicado (o de su entrada estándar).

mkdir (*make dir*)

crea un directorio.

mkfifo

crea una tubería con nombre ligado al sistema de archivos.

mv (*move*)

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

nl (*number lines*)

lee líneas de un archivo 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 archivo 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 archivo en trozos del tamaño indicado.

stat produce información detallada de archivos.

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 archivo indicado como argumento. Con la opción **-f/--follow** monitoriza el archivo mostrando el nuevo contenido tan pronto como se añade. Muy útil para hacer el seguimiento de un archivo de log.

tee crea una «te». Lee de su entrada estándar y lo escribe al mismo tiempo a su salida estándar y al archivo indicado.

test, [¹

realiza comprobaciones lógicas sobre archivos, cadenas de texto y datos numéricos. Se utiliza habitualmente como condición en estructuras de control **if**, **while**, etc.

touch

cambia la fecha de un archivo (por defecto ahora). Si el archivo indicado no existe, crea uno vacío.

¹No es una errata: [, el corchete de apertura

tr (*translate*)

lee líneas de su entrada y las escribe a su salida reemplazando *tokens*² 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 archivo 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:

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

A.2. Sistema

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

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

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

fdisk

muestra y manipula tablas de particiones.

mount

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

uname

(*unix name*) muestra información del sistema: núcleo, hostname, versión, arquitectura, etc.

```
alice@maine:~$ uname -a
Linux maine 6.12.38+deb13-amd64 #1 SMP Debian 6.12.38-1 (2025-07-16) x86_64
↪ GNU/Linux
```

sync escribe a disco inmediatamente las operaciones pendientes sobre archivos.

²Un *token* es cualquier combinación de caracteres que cumpla una expresión regular concreta.

A.3. Procesos

nice ejecuta un programa fijando un nivel de prioridad.

nohup

ejecuta un comando ignorando las señales para su finalización (SIGHUP).
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.

A.4. Usuarios y permisos

chmod (*change mode*)

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

chmod (*change owner*)

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

chgrp (*change group*)

cambia el grupo propietario de un archivo.

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.

Anexo B

Otros comandos de red

B.1. Otros *ping*

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

`fping`

Permite especificar múltiples destinos que serán comprobados secuencialmente.

`oping`

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

B.2. `mtr`

`mtr` es muy similar a `traceroute`, pero puede utilizarse con interfaz gráfica, tal como se muestra en la Figura B.1.

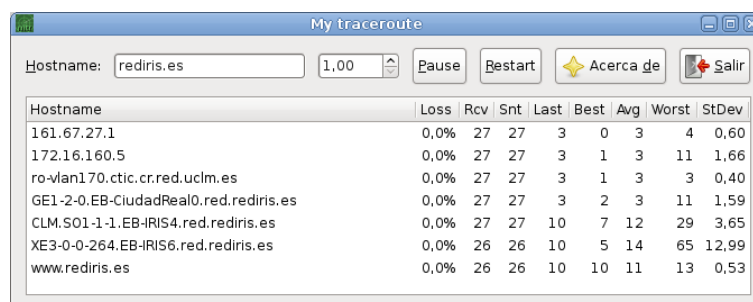


FIGURA B.1: Aspecto del programa `mtr`

B.3. ss

El comando `ss` proporciona mucha información sobre el estado de las conexiones TCP del propio nodo. Puedes utilizar el ejemplo que vimos en la § 12.6 para analizar la información que ofrece. El Listado B.1 muestra una parte a modo de ejemplo.

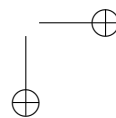
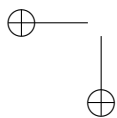
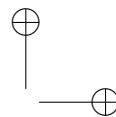
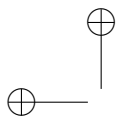
```
~$ ss -timon
State Recv-Q Send-Q Local Address:Port Peer Address:Port
ESTAB 0      2150144 127.0.0.1:33188    127.0.0.1:2000    timer:(persist,348ms,0)
^I skmem:(r0,rb131072,t0,tb2626560,f2176,w2185088,o0,bl0,d0) ts sack cubic wscale:7,7
↪ rto:220 backoff:1 rtt:17.255/0.045 mss:54976 pmtu:65535 rcvmss:536 advmss:65483
↪ cwnd:10 bytes_sent:1894649664 bytes_retrans:164928 bytes_acked:1894484737
↪ segs_out:51699 segs_in:51693 data_segs_out:34467 send 254887279bps lastsnd:332
↪ lastrcv:9472192 lastack:92 pacing_rate 509767168bps delivery_rate 50712936bps
↪ delivered:34468 busy:947218ms rwnd_limited:9472180ms(100.0%) retrans:0/3
↪ dsack_dups:3 rcv_space:65495 rcv_ssthresh:65495 notsent:2150144 minrtt:12.982
↪ rcv_wnd:65536
```

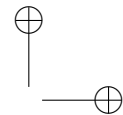
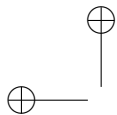
LISTADO B.1: TCP: Análisis de una conexión con el programa `ss`

Veamos los datos que ofrece:

- **State:** Estado de la conexión.
- **Recv-Q y Send-Q:** Datos pendientes en los buffers de recepción y envío respectivamente.
- **Local Address:Port y Peer Address:Port:** Direcciones y puertos de los extremos de la conexión.
- **skmem:** Información sobre la memoria del socket (socket memory):
 - **r:** Memoria ocupada en el buffer de recepción (bytes).
 - **rb:** Memoria total en el buffer de recepción (bytes).
 - **t:** Memoria ocupada en el buffer de envío (bytes).
 - **tb:** Memoria total en el buffer de envío (bytes).
 - **f:** Memoria utilizada como caché (bytes).
 - **w:** Memoria ocupada por paquetes encolados (bytes).
 - **o:** Memoria ocupada por opciones del socket (bytes).
 - **bl:** Memoria ocupada por el backlog (bytes).
 - **d:** Número de paquetes descartados.
- **ts:** La conexión está usando timestamps para medir el RTT.
- **sack:** La confirmación selectiva está habilitada.
- **cubic:** Algoritmo de control de congestión.
- **wscale:<envío>,<recepción>:** Factor de escala de las ventanas.
- **rto:** RTO actual (ms).
- **rtt:<medio>/<dev>:** RTT medio y desviación estándar (ms).
- **ato:** Auto TimeOut (ms) antes de enviar un ACK.
- **mss:** MSS (bytes).
- **pmtu:** Path MTU (bytes).
- **rcvmss:** MSS de recepción (bytes).
- **advmss:** MSS anunciado por el receptor (bytes).
- **cwnd:** Ventana de congestión (en segmentos).

- `bytes_sent`: Bytes enviados.
- `bytes_retrans`: Bytes retransmitidos.
- `bytes_acked`: Bytes confirmados.
- `bytes_received`: Bytes recibidos.
- `segs_out`: Segmentos enviados.
- `segs_in`: Segmentos recibidos.
- `data_segs_out`: Segmentos enviados que contienen datos.
- `data_segs_in`: Segmentos recibidos que contienen datos.
- `send`: Velocidad de envío.
- `lastsnd`: Último segmento enviado (ms desde el inicio).
- `lastrcv`: Última vez que se recibió un segmento (ms desde el inicio).
- `lastack`: Última vez que se recibió un ACK.
- `pacing_rate`: Velocidad de envío para evitar congestión.
- `delivery_rate`: Tasa de entrega efectiva.
- `delivered`: Número de segmentos entregados con éxito.
- `app_limited`: La tasa está limitada por la aplicación, no por la red.
- `busy`: Tiempo total de la conexión.
- `retrans`: Número de retransmisiones.
- `dsack_dups`: Número de segmentos duplicados.
- `rcv_rtt`: RTT de recepción estimado (ms).
- `rcv_space`: Tamaño del buffer de recepción.
- `rcv_ssthresh`: `ssthresh` (bytes).
- `minrtt`: RTT mínimo observado (ms).
- `snd_wnd`: Ventana de envío (bytes).
- `rcv_wnd`: Ventana de recepción (bytes).





Anexo C

Docker

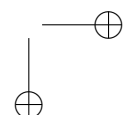
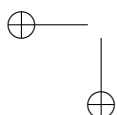
A lo largo del libro se han incluido varios ejemplos que requieren cierta infraestructura y que hemos llamado ‘laboratorios’. Para implementarlos hemos utilizado `docker`, una herramienta que permite ejecutar aplicaciones dentro de contenedores. Un contenedor es un entorno de ejecución aislado, separado del SO anfitrión que tiene su propio sistema de archivos, configuración de red y procesos. Esto nos permite simular varios nodos que se comunican entre sí, obviamente muy útil cuando hablamos de redes, aunque no sea ese su propósito principal.

Al empezar a utilizar `docker` puede recordar bastante a sistemas de VM (Virtual Machine) como `VirtualBox` o `VMware`, sin embargo, hay diferencias importantes:

- El contenedor `docker` no emula el hardware de un computador, ni ejecuta un SO completo, sino que utiliza los programas sobre el SO del anfitrión. No permite ejecutar un SO Windows sobre un SO GNU/Linux o viceversa, ni siquiera de una versión diferente del núcleo.
- A diferencia de las VM, un contenedor `docker` está pensado para ejecutar un único proceso. Cuando el proceso termina, el contenedor también. Esto hace que la imagen del contenedor sea normalmente mucho más pequeña que la de una VM.
- Los contenedores son efímeros, es decir, no tienen estado. Cualquier cambio (escritura) que se aplique al sistema de archivos del contenedor se pierde al pararlo, aunque es posible montar volúmenes externos persistentes.

C.1. Imágenes

Para ejecutar un contenedor `docker` necesitas una imagen. La imagen la puedes obtener de un repositorio de imágenes o *hub* (como <https://hub.docker.com>) o bien la puedes crearla tu mismo a partir de otra.



La prueba habitual para comprobar que tienes una instalación correcta y funcional es ejecutar un contenedor con la imagen `hello-world`, que siendo la primera vez, conllevará la descarga automática de dicha imagen.

```

1 ~$ docker run hello-world
2 Unable to find image 'hello-world:latest' locally
3 latest: Pulling from library/hello-world
4 17eec7bbc9d7: Pull complete
5 Digest: sha256:1d983076c7facf19a9a53a7a4855dbd736bf5206e3bfc5142a4c1da4ed44dedc
6 Status: Downloaded newer image for hello-world:latest
7
8 Hello from Docker!
9 This message shows that your installation appears to be working correctly.
10
11 To generate this message, Docker took the following steps:
12 1. The Docker client contacted the Docker daemon.
13 2. The Docker daemon pulled the "hello-world" image from the Docker Hub.
14    (amd64)
15 3. The Docker daemon created a new container from that image which runs the
16    executable that produces the output you are currently reading.
17 4. The Docker daemon streamed that output to the Docker client, which sent it
18    to your terminal.
19
20 To try something more ambitious, you can run an Ubuntu container with:
21 $ docker run -it ubuntu bash
22
23 Share images, automate workflows, and more with a free Docker ID:
24 https://hub.docker.com/
25
26 For more examples and ideas, visit:
27 https://docs.docker.com/get-started/

```

La **línea 2** dice que la imagen solicitada no está en tu computador, por lo que procede a descargarla. Las **líneas 3-6** muestran información sobre la descarga. Finalmente, a partir de la **línea 7** aparece la salida del programa `hello` que es un programa incluido en la imagen y que se ejecuta dentro del contenedor.

Cualquier imagen descargada se almacena en tu computador y se puede reutilizar. Puedes comprobarlo con el siguiente comando:

```

~$ docker images

```

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
hello-world	latest	1b44b5a3e06a	6 days ago	10.1kB

C.2. Contenedores

El contenedor representa la ejecución del programa definido en una imagen, y de hecho puedes crear varios contenedores sobre la misma imagen. Puedes probar un contenedor que ejecuta un servidor web (`nginx`). De hecho eso, es lo habitual: `docker` se suele usar para ejecutar un servidor o un servicio que queda en ejecución indefinidamente.

```
~$ docker run -d -p 8000:80 -v ./data nginx
fc085a4e035a6d07d0ab22ed6d03887d99a0ad3c6ff84cc221ee6b4eb09dbac5
```

Los argumentos de ese comando son los más frecuentes:

- **-d**: ejecuta el contenedor en segundo plano. La secuencia alfanumérica que ves en este caso es el identificador del contenedor.
- **-p 8000:80**: consigue que el puerto TCP 80 del contenedor esté accesible a través del 8000 de tu computador y con ello dar acceso al servicio desde el exterior. Puedes probar a acceder a ese servidor web cargando <http://localhost:8000> en tu navegador y verás la página de bienvenida de nginx.
- **-v ./data**: monta el directorio actual (filename.) de tu computador en el directorio `/data` del contenedor. Eso da al contenedor acceso a los archivos de ese directorio y además le otorga persistencia.
- **nginx**: es el nombre de la imagen, de forma análoga al `hello-world` del ejemplo anterior.

Puedes comprobar el estado de los contenedores y sus procesos asociados con `docker ps` o `docker container ls`, aunque este segundo comando muestra también los contenedores parados.

```
~$ docker container ls
```

CONTAINER ID	IMAGE	COMMAND	PORTS	NAMES
fc085a4e035a	nginx	"/docker-entrypoint..."	0.0.0.0:8000->80/tcp	hardcore_cannon

En esta sencilla información de estado del contenedor aparece el comando que se ejecuta en el interior del contenedor (llamado menudo *entrypoint*), el puerto mapeado 8000->80 y el nombre del contenedor (`hardcore_cannon`) que se genera aleatoriamente y no debes confundir con el nombre de la imagen.

El contenedor seguirá en ejecución indefinidamente hasta que lo detengas u ocurra un error grave. En esta situación `docker` te permite ejecutar otros comandos del contenedor con la orden `docker exec`. Por ejemplo, esto ejecuta una shell interactiva (gracias a la opción `-ti`) dentro del contenedor:

```
~$ docker exec -ti hardcore_cannon bash
root@1ad721a249fe:/#
```

Por último puedes parar un contenedor activo con:

```
~$ docker stop hardcore_cannon
hardcore_cannon
```

C.3. Dockerfile

Si quieres crear una imagen propia, o mejor dicho, personalizar una imagen existente puedes crear un archivo `Dockerfile`. Por ejemplo, supongamos que necesitas tener Python disponible en el contenedor anterior. Puedes crear un `Dockerfile` con lo siguiente:

```
FROM nginx
RUN apt-get update && apt-get install -y python3
EXPOSE 80
```

La primera línea es la imagen base, la segunda es la ejecución de dos comandos para instalar el paquete `python3` y la tercera informa de que el contenedor expone el puerto 80. Para poder usar esta imagen, primero debes construirla:

```
~$ docker build -t nginx-python .
[+] Building 11.3s (6/6)FINISHED                                docker:default
=> [internal] load build definition from Dockerfile              0.0s
=> => transferring dockerfile: 107B                             0.0s
=> [internal] load metadata for docker.io/library/nginx:latest  0.0s
=> [internal] load .dockerignore                                0.0s
=> => transferring context: 2B                                   0.0s
=> [1/2] FROM docker.io/library/nginx:latest                   0.1s
=> [2/2] RUN apt-get update && apt-get install -y python3       10.2s
=> exporting to image                                           0.9s
=> => exporting layers                                          0.8s
=> => writing image sha256:b1141380ef52995902f0c568a82feb4fcfe122d8a... 0.0s
=> => naming to docker.io/library/nginx-python                 0.0s
```

El parámetro `-t` asigna un nombre a la imagen y el punto final se refiere al directorio actual, que es donde debe buscar el archivo `Dockerfile`. Ahora la nueva imagen aparecerá en la lista:

```
~$ docker images
nginx-python      latest      b1141380ef52  2 minutes ago  248MB
nginx             latest      ad5708199ec7  44 hours ago   192MB
hello-world       latest      1b44b5a3e06a  6 days ago     10.1kB
```

Y con esto puede arrancar un contenedor con la imagen recién creada:

```
~$ docker run -d -p 8000:80 nginx-python
```

C.4. Docker Compose

Cuando se necesitan varios contenedores que deben cooperar o comunicarse entre sí, es mucho más cómodo utilizar `docker-compose`. Este programa lee un archivo llamado `docker-compose.yml` que indica la configuración de varios contenedores y sus relaciones.


```
services:
  nginx:
    image: nginx
    container_name: web-server
    ports:
      - "8080:80"
    volumes:
      - ./html:/usr/share/nginx/html:ro
    depends_on:
      - db

  db:
    image: postgres
    container_name: db
    environment:
      POSTGRES_USER: usuario
      POSTGRES_PASSWORD: password
      POSTGRES_DB: storage
    ports:
      - "5432:5432"
    volumes:
      - db_data:/var/lib/postgresql/data

volumes:
  db_data:
```

Este archivo define dos servicios (contenedores). Veámoslos en detalle:

- **nginx** es el servidor web, equivalente al del ejemplo de la sección anterior.
 - **ports** define redirecciones de puertos como el parámetro **-p** en el comando **run**.
 - **volumes** define un punto de montaje, como la opción **-p** aunque es este caso de indica que es solo lectura (**ro**), de modo que el contenedor no podrá modificar el contenido de este directorio.
 - **depends_on** dice que el contenedor **db** debería arrancar antes que este.
- **db** ejecuta la base de datos PostgreSQL (versión 15).
 - **environment** define variables de entorno. En este caso se define el nombre de la base de datos y el nombre de usuario y clave con el que se accede. Las variables de entorno son la forma más habitual de configurar servicios **docker**.
 - **volumes** en este caso especifica un volumen gestionado por **docker**. Fíjate que la parte antes del **:** no es una ruta. Es un identificador que aparece al final en la sección **volumes**.

Para arrancar los contenedores basta con ejecutar el siguiente comando en el directorio dónde está **docker-compose.yml**:

```
~$ docker-compose up -d
[+] Running 4/4
- Network compose-example_default Created 0.0s
- Volume "compose-example_db_data" Created 0.0s
- Container db Started 0.3s
- Container web-server Started 0.4s
```

Además, `docker` crea automáticamente una red interna con direccionamiento privado a la que conecta todos los contenedores. El nombre de la red se basa en el directorio que contiene el archivo `docker-compose.yml`. Puedes obtener información de las redes disponibles con:

```
~$ docker network ls
NETWORK ID      NAME                                DRIVER  SCOPE
32ef38990d8e    bridge                             bridge  local
dc3b83ef6011    compose-example_default            bridge  local
059b37dd8ae6    host                               host    local
030f578f0ff0    none                               null    local
```

Y puedes ver detalles sobre esa red y las direcciones asignadas a los contenedores:

```
~$ docker network inspect compose-example_default
[...]
```

```
  "IPAM": {"Config": [{
    "Subnet": "172.19.0.0/16",
    "Gateway": "172.19.0.1"
  }]},
[...]
```

```
  "Containers": {
    "3748abed03d58138129ee0f17d6f1a5a9a4df22584fef184fd1d7968f8c7aeed": {
      "Name": "db",
      "MacAddress": "c2:d7:a3:24:c0:b5",
      "IPv4Address": "172.19.0.2/16",
    },
    "5e6cb60767f0b5e00ba8e60eec9692dd073da56826af27ea3cc8db659d3151e6": {
      "Name": "web-server",
      "MacAddress": "86:47:2d:6b:5a:20",
      "IPv4Address": "172.19.0.3/16",
    }
  },
[...]
```

Anexo D

Unidades

Una pequeña referencia de las unidades de medida más comunes en redes y comunicaciones, que por supuesto se utilizan en este libro.

D.1. Memoria

La unidad mínima de memoria es el byte (que se abrevia como **B** mayúscula) y para sus múltiplos se aplican los prefijos del SI (Sistema de Información) para potencias de 2, es decir:

múltiplo	prefijo	relación	en bytes
kibibyte	KiB	1 024 B	2^{10} B
mebibyte	MiB	1 024 KiB	2^{20} B
gibibyte	GiB	1 024 MiB	2^{30} B
tebibyte	TiB	1 024 GiB	2^{40} B
pebibyte	PiB	1 024 TiB	2^{50} B

CUADRO D.1: Múltiplos (binarios) de byte como medida de memoria.

D.2. Almacenamiento

El almacenamiento también se mide en bytes, pero se utilizan los prefijos del SI para potencias de 10, es decir:

múltiplo	prefijo	relación	en bytes
kilobyte	kB	1 000 B	10^3 B
megabyte	MB	1 000 kB	10^6 B
gigabyte	GB	1 000 MB	10^9 B
terabyte	TB	1 000 GB	10^{12} B
petabyte	PB	1 000 TB	10^{15} B

CUADRO D.2: Múltiplos (decimales) de byte como medida de almacenamiento

Fíjate que el prefijo de ‘kilo’ se escribe con *k* minúscula mientras que para ‘kibi’ se escribe con *K* mayúscula. También es importante tener claro que en muchos textos y especificaciones técnicas se asumen potencias de 2 aunque se utilicen los prefijos decimales si se refieren a cantidades de memoria.

D.3. Tasa o velocidad de transmisión

Lo más habitual es medir la velocidad de transmisión (*data rate*) en múltiplos decimales del ‘bit por segundo’, denotado como bit/s¹, b/s o más comúnmente bps.

prefijo	unidad SI	relación	en bps
kbps	kbit/s	1 000 bps	10 ³ bps
Mbps	Mbit/s	1 000 kbps	10 ⁶ bps
Gbps	Gbit/s	1 000 Mbps	10 ⁹ bps
Tbps	Tbit/s	1 000 Gbps	10 ¹² bps
Pbps	Pbit/s	1 000 Tbps	10 ¹⁵ bps

CUADRO D.3: Múltiplos de bps.

Aunque menos común, también se emplean múltiplos decimales del ‘byte por segundo’, denotado como B/s.

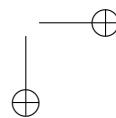
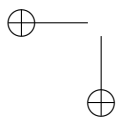
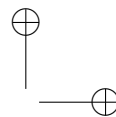
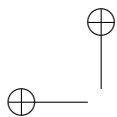
prefijo	relación	en B/s
kB/s	1 000 B/s	10 ³ B/s
MB/s	1 000 kB/s	10 ⁶ B/s
GB/s	1 000 MB/s	10 ⁹ B/s
TB/s	1 000 GB/s	10 ¹² B/s
PB/s	1 000 TB/s	10 ¹⁵ B/s

CUADRO D.4: Múltiplos de B/s.

Es común asimilar el concepto de velocidad (o tasa) de transmisión con el *ancho de banda*. Aunque puede servir para tener una idea de la capacidad de un canal, no son lo mismo. El ancho de banda, como su nombre indica, define la anchura de la banda de frecuencias (medidas en hercios) que utiliza un canal para transportar señales que codifican datos —digitales para el caso que nos ocupa. Por ejemplo, un canal que transmita entre los 2 kHz y los 5 kHz tiene un ancho de banda de 3 kHz. Pero la velocidad de transmisión depende del sistema de codificación. Si el sistema solo permite codificar 1 bit por ciclo, en ese canal solo se puede transmitir datos a 3 kbps, en cambio un sistema como 16-QAM codifica 4 bits por ciclo, de modo que

¹bit/s es lo correcto según el SI.

permitiría una velocidad de 12 kbps para el mismo ancho de banda de 3 kHz.

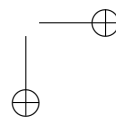
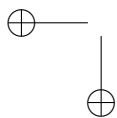
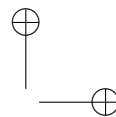
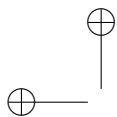


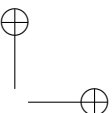
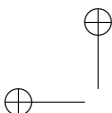
Índice alfabético

- 3dupack, 249
- accept(), 92, 93
- Acceptor, 92
- almacenamiento y reenvío, 130
- aplicación distribuida, 259
- AQM, 253
- ARP, 69
- asyncio, 283
- BGP, 177
- CIDR, 120
- conectividad, 54
- conmutación de paquetes, 131
- connect(), 94
- dig, 380
- Dijkstra (algoritmo), 170
- DNS autoritativo, 382
- Don't fragment, 161
- dupack, 249
- E/S parcial, 95
- ECN, 253
- ECT, 253
- EGP, 177
- encaminamiento
 - algoritmo, 166
 - dinámico, 165
 - estático, 165
 - protocolo, 166
- encapsulación, 54
- endianness, *véase* ordenamiento de bytes
- endpoint, 87
- entrega directa, 133
- entrega indirecta, 133
- Ethernet, 59
- FQDN, 382
- fragmentación, 152
- glue record, 391
- go back N, *véase* repetición continua
- ICMP, 72, 143
 - Address mask, 152
 - Destination unreachable, 145
 - Echo, 149
 - Information, 152
 - Parameter problem, 148
 - Redirect, 148
 - Router advertisement, 152
 - Router solicitation, 152
 - Time exceeded, 147
 - Timestamp, 151
- Identificador de proceso, 13
- internet, 43
- interred, 43
- IP, 65
- Jacobson (algoritmo), 226

- Karn (algoritmo), 227
- keep-alive (temporizador), 237
- kill (comando), 13
- listen(), 93
- listener socket, 93
- loopback, 66
- MAC, 61
- modelo híbrido, 50
- MQTT, 305
- MSS, 214
- mtr, appmtr411
- MTU, 152
- MTU path discovery, 162
- MTU path negotiation, 162
- multiplexación, 46
- máscara de red, 116
- Nagle (algoritmo), 229
- NAPT, 375
- NAT, 374
- ncat, 108
- netcat, 108
- netmask, *véase* máscara de red
- NIC, 59
- ordenamiento de bytes, 328
- OSI (modelo), 48
- parada y espera, 205
- partial I/O*, *véase* E/S parcial
- persistencia (temporizador), 230
- PID, 13
- ping, 75
- preforking*, 269
- puerto, 77, 88, 365
- puerto 0, 90
- puerto abierto, 89
- quiet time, *véase* tempo de silencio218
- recv(), 92
- recvfrom(), 91
- RED, 253
- red de computadores, 43
- red de datagramas, 131
- repetición continua, 206
- repetición selectiva, 208
- retransmisión (temporizador), 224
- router, 130
- RTO, 224
- RTT, 75, 224
- segmento, 80
- select(), 281, 299
- selective repeat, *véase* repetición selectiva
- selectors, 283, 300
- send(), 92
- sendall(), 96
- sending buffer (TCP), 96
- send(), 96
- sendto(), 89
- silly window, *véase* ventana tonta
- sistema autónomo, 167
- sniffer, 337
- SO_REUSEADDR, 218
- sobrecarga de cabeceras, 214
- socket, 86
- socket.fileno(), 101
- socket.makefile(), 102
- socket.recv(), 97
- socketserver, 277
- stall*, 315
- stop and wait, *véase* parada y espera

- store and forward, *véase*
 - almacenamiento y reenvío
- tabla de encaminamiento, 134
- TCP, 79
 - ACK, 211
 - cliente, 263
 - confiabilidad, 211
 - confirmación duplicada, 236
 - FIN, 216
 - flags, 81
 - mensaje, 80
 - RST, 218
 - SACK, 230
 - servidor, 261
 - servidor multihilo, 265
 - servidor multiproceso, 267
 - SYN, 213
- TCP/IP (modelo), 49
- tiempo de silencio, 218
- TLD, 382
- traceroute, `traceroute`147
- tshark, 55, 338
- TTL, 132
- UDP, 76
 - mensaje, 77
 - servidor, 271
- UDP, 89
- vector-ruta, 177
- ventana tonta, 229
- wireshark, 348





Referencias

- [1] “Debian GNU/Linux.” <http://www.debian.org/>.
- [2] D. M. Ritchie and B. W. Kernighan, *The C Programming Language*. Englewood Cliffs, NJ: Prentice Hall, 1978.
- [3] B. Gates, “An open letter to hobbyists.” Carta abierta distribuida en el entorno del Homebrew Computer Club, Feb. 1976. https://www.digibarn.com/collections/newsletters/homebrew/V2_01/gatesletter.html.
- [4] R. Stallman, “The gnu manifesto,” 1985. <https://www.gnu.org/gnu/manifesto.en.html>.
- [5] L. Torvalds, “What would you like to see most in minix?.” Mensaje en el grupo de noticias comp.os.minix, Aug. 1991. <https://web.archive.org/web/20130509134305/http://groups.google.com/group/comp.os.minix/msg/b813d52cbc5a044b>.
- [6] I. Murdock, “Debian announcement.” Mensaje en el grupo de noticias comp.os.linux, Aug. 1993. <https://groups.google.com/g/comp.os.linux.development/c/Md3Modzg5TU/m/xy88y50LaMJ>.
- [7] L. Delgrossi and L. Berger, “Internet Stream Protocol Version 2 (ST2) Protocol Specification - Version ST2+.” RFC 1819 (Experimental), Aug. 1995.
- [8] W. Simpson, “The Point-to-Point Protocol (PPP).” RFC 1661 (Standard), July 1994. Updated by RFC 2153.
- [9] J. Postel, “Internet Protocol.” RFC 791 (Standard), Sept. 1981. Updated by RFC 1349.
- [10] W. Fenner, “Internet Group Management Protocol, Version 2.” RFC 2236 (Proposed Standard), Nov. 1997. Obsoleted by RFC 3376.
- [11] J. Postel, “Internet Control Message Protocol.” RFC 792 (Standard), Sept. 1981. Updated by RFCs 950, 4884.

- [12] T. Pummill and B. Manning, “Variable Length Subnet Table For IPv4.” RFC 1878 (Historic), Dec. 1995.
- [13] V. Fuller, T. Li, J. Yu, and K. Varadhan, “Classless Inter-Domain Routing (CIDR): an Address Assignment and Aggregation Strategy.” RFC 1519 (Proposed Standard), Sept. 1993. Obsoleted by RFC 4632.
- [14] A. Retana, R. White, V. Fuller, and D. McPherson, “Using 31-Bit Prefixes on IPv4 Point-to-Point Links.” RFC 3021 (Proposed Standard), Dec. 2000.
- [15] F. Gont, “Deprecation of ICMP Source Quench Messages.” RFC 6633, May 2012.
- [16] F. Gont and C. Pignataro, “Formally Deprecating Some ICMPv4 Message Types.” RFC 6918, Apr. 2013.
- [17] S. Deering, “ICMP Router Discovery Messages.” RFC 1256 (Proposed Standard), Sept. 1991.
- [18] E. W. Dijkstra, “A note on two problems in connection with graphs,” *Numerische Mathematik*, vol. 1, no. 1, pp. 269–271, 1959.
- [19] C. L. Hedrick, “Routing Information Protocol.” RFC 1058 (Historic), June 1988. Updated by RFCs 1388, 1723.
- [20] G. Malkin, “RIP Version 2.” RFC 2453 (Standard), Nov. 1998. Updated by RFC 4822.
- [21] J. Moy, “OSPF Version 2.” RFC 1247 (Draft Standard), July 1991. Obsoleted by RFC 1583, updated by RFC 1349.
- [22] R. Droms, “Dynamic Host Configuration Protocol.” RFC 2131 (Draft Standard), Mar. 1997. Updated by RFCs 3396, 4361.
- [23] W. Eddy, “Transmission Control Protocol (TCP).” RFC 9293, Aug. 2022.
- [24] J. Postel, “Transmission Control Protocol.” RFC 793 (Standard), Sept. 1981. Updated by RFCs 1122, 3168.
- [25] M. Sargent, J. Chu, D. V. Paxson, and M. Allman, “Computing TCP’s Retransmission Timer.” RFC 6298, June 2011.
- [26] V. Jacobson, “Congestion avoidance and control,” in *Proceedings of the ACM SIGCOMM ’88 Conference on Communications Architectures and Protocols*, p. 314–329, ACM, 1988.

- [27] P. Karn and C. Partridge, “Improving round-trip time estimates in reliable transport protocols,” *ACM SIGCOMM Computer Communication Review*, vol. 17, no. 5, p. 2–7, 1987.
- [28] J. Nagle, “Congestion control in IP/TCP internetworks.” RFC 896, Jan. 1984.
- [29] R. Braden, “Requirements for Internet Hosts - Communication Layers.” RFC 1122 (Standard), Oct. 1989. Updated by RFCs 1349, 4379.
- [30] F. Baker, “Requirements for IP Version 4 Routers.” RFC 1812 (Proposed Standard), June 1995. Updated by RFC 2644.
- [31] K. Ramakrishnan, S. Floyd, and D. Black, “The Addition of Explicit Congestion Notification (ECN) to IP.” RFC 3168 (Proposed Standard), Sept. 2001.
- [32] J. Iyengar and M. Thomson, “QUIC: A UDP-Based Multiplexed and Secure Transport.” RFC 9000, May 2021.
- [33] J. Reynolds and J. Postel, “Assigned Numbers.” RFC 1700 (Historic), Oct. 1994. Obsoleted by RFC 3232.
- [34] Y. Rekhter, B. Moskowitz, D. Karrenberg, G. J. de Groot, and E. Lear, “Address Allocation for Private Internets.” RFC 1918 (Best Current Practice), Feb. 1996.
- [35] P. Srisuresh and K. Egevang, “Traditional IP Network Address Translator (Traditional NAT).” RFC 3022 (Informational), Jan. 2001.
- [36] P. Srisuresh and M. Holdrege, “IP Network Address Translator (NAT) Terminology and Considerations.” RFC 2663 (Informational), Aug. 1999.
- [37] P. V. Mockapetris, “Domain names - concepts and facilities.” RFC 1034 (Standard), Nov. 1987. Updated by RFCs 1101, 1183, 1348, 1876, 1982, 2065, 2181, 2308, 2535, 4033, 4034, 4035, 4343, 4035, 4592.
- [38] P. Vixie, S. Thomson, Y. Rekhter, and J. Bound, “Dynamic Updates in the Domain Name System (DNS UPDATE).” RFC 2136 (Proposed Standard), Apr. 1997. Updated by RFCs 3007, 4035, 4033, 4034.