MÁSTER EN REVERSING, ANÁLISIS DE MALWARE Y BUG HUNTING

MÓDULO 5. REVERSING DE REDES Y PROTOCOLO

# MÁSTER EN *ANÁLISIS DE MALWARE Y REVERSING*

Campus Internacional CIBERSEGURIDAD

ENIIT INNOVA IT BUSINESS SCHOOL

UCAM UNIVERSIDAD CATÓLICA DE MURCIA

# Contenido

# 1. Introducción

El objetivo de este curso es proporcionar los conocimientos, técnicas y experiencia necesarias para ser capaz de analizar el tráfico generado por una aplicación de red, y de esa forma ser capaz de entender su funcionamiento y el protocolo que emplea para comunicarse con otras entidades.

## 1.1. Requisitos previos

Aunque es recomendable tener ciertas nociones sobre las redes TCP/IP, el curso intenta explicar los protocolos más importantes de la pila TCP/IP, al menos en el contexto de entender su tráfico y las aplicaciones que lo generan.

También se recomienda ser capaz de interpretar código C y cierta experiencia en el uso de la línea de comandos de los sistemas operativos Linux/UNIX, para poder ejecutar las herramientas de captura y análisis de tráfico propuestas. Aunque la mayoría de ellas son multiplataforma, por lo que pueden ejecutarse en el sistema operativo preferido.

## 1.2. Estructura del curso

Este curso está estructurado en 6 temas:

1. **Repaso de TCP/IP**. En este capítulo repasaremos la torre de protocolos TCP/IP que emplean las aplicaciones que se comunican a través de Internet. Después de entender cómo funciona una torre de protocolos, como encapsula y envía información, y los principales elementos de interconexión que se emplean en Internet, reparásemos los protocolos de comunicación más populares hoy en día: Ethernet, IPv4, ARP, ICMP, UDP, TCP, DNS, TLS y HTTP.

2. **Captura y Análisis de Tráfico de Red**. Una vez que entendamos cómo funcionan los principales protocolos de Internet, estudiaremos las diferentes estrategias que existen para capturar el tráfico que generan las aplicaciones de red, y analizaremos las tres principales herramientas que usaremos en este curso: 'tcpdump', para capturar tráfico a nivel de red, 'Wireshark' para analizar el tráfico capturado, y 'mitmproxy' que nos permite capturar y analizar tráfico a nivel de aplicación.

3. **Análisis de Aplicaciones basadas en el protocolo UDP**. En este capítulo empezaremos a analizar las aplicaciones basadas en UDP, el protocolo de transporte más sencillo. Empezaremos estudiando el código fuente de una aplicación cliente-servidor de UDP de ejemplo, las llamadas al sistema y funciones de red más importantes que emplea y por último analizaremos su tráfico de red.

4. **Análisis de Aplicaciones basadas en el protocolo TCP**. En este capítulo estudiaremos las aplicaciones basadas en TCP, el protocolo de transporte más importante de Internet. Este capítulo tiene la misma estructura del anterior, y en él se analizará una aplicación cliente-servidor TCP de ejemplo, las llamadas al sistema específicas de TCP y el tráfico de red que generan las aplicaciones TCP.

5. **Análisis de aplicaciones basadas en *sockets* "crudos"**. Aunque la mayoría de las aplicaciones emplean TCP o UDP, hay ciertas aplicaciones que requieren implementar otros protocolos de la pila de protocolos TCP/IP, usando los que se conoce como sockets "crudos" (*raw sockets*, en inglés). En este capítulo analizaremos una aplicación ICMP de ejemplo, estudiando las llamadas a sistema que emplea, y cómo es el tráfico que genera.

6. **Captura y Análisis de Tráfico TLS/SSL**. Este capítulo describe las técnicas existentes para intentar descifrar o interceptar el tráfico de las aplicaciones que emplean TLS, el protocolo seguro más importante de hoy en día. Estudiaremos tanto técnicas pasivas de descifrado, como los ataques de *Adversary-in-the-Middle* (AitM) contra TLS, así como sus limitaciones.

   **Bibliografía adicional**. Un listado de libros de interés para estudiar en más profundidad los contenidos de este curso.

# 2. Repaso a la Torre de Protocolos TCP/IP

En este capítulo repasáremos la torre TCP/IP y los principales protocolos que la forman.

## 2.1. La Torre de Protocolos TCP/IP

Gracias a Internet, nos hemos acostumbrado a poder enviar información entre cualquier punto de mundo de manera fiable y casi instantánea. Pero para conseguir esta proeza tecnológica es necesario resolver un gran número de problemas, demasiado complejos como para poder resueltos con una única tecnología. Por esa razón, las comunicaciones a través de Internet utilizan lo que se conoce como pila de protocolos, en la que cada uno de los protocolos se encarga de resolver una pequeña parte de la comunicación.

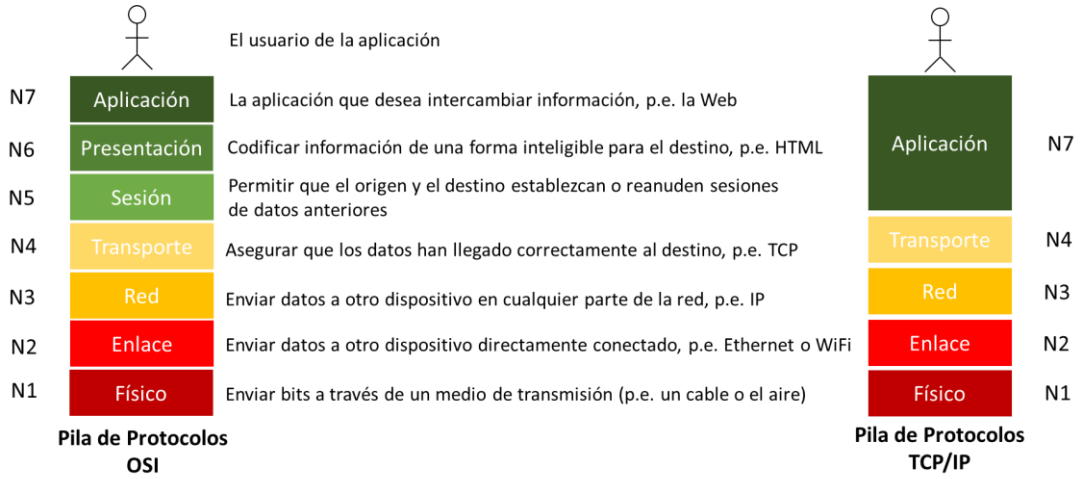



**Figura 2.1** – Las torres de protocolos OSI y TCP/IP

A lo largo de la historia se han propuesto varias torres de protocolos, como la torre OSI que se divide en 7 capas o niveles, cada una con una función diferente (**Figura 2.1**). La torre de protocolos que se usa hoy en día en Internet se denomina TCP/IP y, aunque tiene menos capas que la torre OSI, es conceptualmente similar.

La idea fundamental de las torres de protocolos es que cada capa utiliza los servicios proporcionados por la capa inmediatamente inferior para comunicarse con la misma capa en el destino (e.g. un protocolo del nivel de transporte utiliza los servicios de un protocolo del nivel de red para comunicarse con el mismo protocolo en el nivel de transporte del destino). El objetivo de esta estructura en capas es ocultar su complejidad, de forma que una capa solo tiene que interactuar con la capa inmediatamente inferior. Normalmente las capas inferiores no saben qué hacen exactamente las capas superiores, ni son capaces de interpretar sus datos, sino que simplemente envían y reciben los datos que éstos les solicitan. Esto

permite reemplazar partes de la pila de protocolos sin impactar más que a la capa inmediatamente superior. Por ejemplo, en una comunicación entre protocolos de transporte se podría utilizar un protocolo de nivel de enlace cableado o uno inalámbrico, sin que el nivel de transporte fuese consciente de la diferencia (más allá del rendimiento que podría tener cada uno).

Como existe un gran número de aplicaciones, y cada una puede tener unos requisitos diferentes respecto a la fiabilidad y latencia de sus comunicaciones, y cada medio de transmisión requiere técnicas de comunicación diferentes, en cada capa de la pila de protocolos pueden existir varios protocolos, que ofrecen servicios diferentes a las capas superiores. Por ejemplo, la torre de protocolos TCP/IP (**Figura 2.2**) tiene un gran número de protocolos de nivel de aplicación, así como de tecnologías de nivel de enlace sobre los que enviar datos, mientras que los protocolos TCP, UDP y IP son los únicos comunes, que aparecen en la gran mayoría de las comunicaciones a través de Internet (de ahí el nombre de TCP/IP).

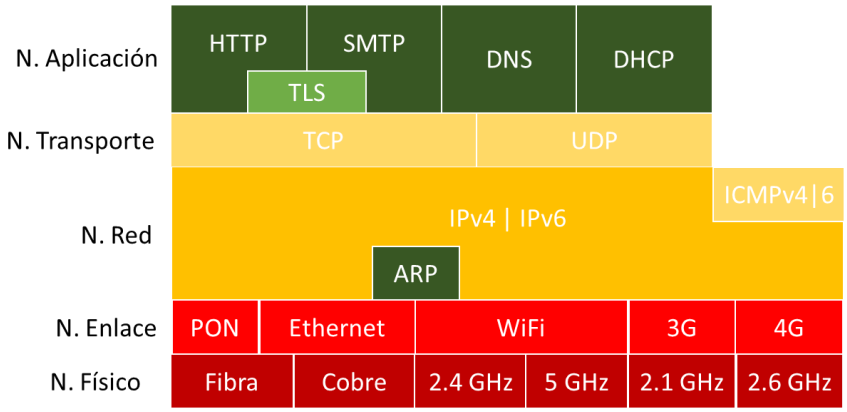



**Figura 2.2** – Protocolos de la torre TCP/IP

¿Pero cómo se comunican exactamente dos ordenadores que implementan una pila de protocolos TCP/IP? Supongamos que, como en la **Figura 2.3**, tenemos un navegador web que quiere enviarle un **mensaje** HTTP (la Word Wide Web se basa en el protocolo HTTP) a un servidor web remoto (y de momento supongamos que están conectados directamente a través de un cable). Para enviarlo, HTTP solicitará a la capa inferior, en este caso TCP, que envíe esos datos al destino. Así que TCP añadirá una cabecera a los datos proporcionados por HTTP (que para TCP no son más que un conjunto de octetos opacos), y entregará este **segmento** (cabecera TCP + mensaje HTTP) a la capa de red para que lo envíe. El protocolo IPv4 repetirá la misma operación: añadirá su cabecera a los datos enviados por TCP y pasará el **datagrama** (cabecera IP + segmento TCP) a Ethernet. Del mismo modo, Ethernet añadirá sus cabeceras a los datos enviados por IP y entregará la **trama** (cabeceras Ethernet + datagrama IP) al nivel físico para que lo envíe. El nivel físico finalmente convertirá esos datos en una señal electromagnética, que se enviarán por el cable

hasta el nivel físico del servidor. Éste recuperará los datos a partir de la señal electromagnética recibida y entregará la trama a la capa Ethernet.

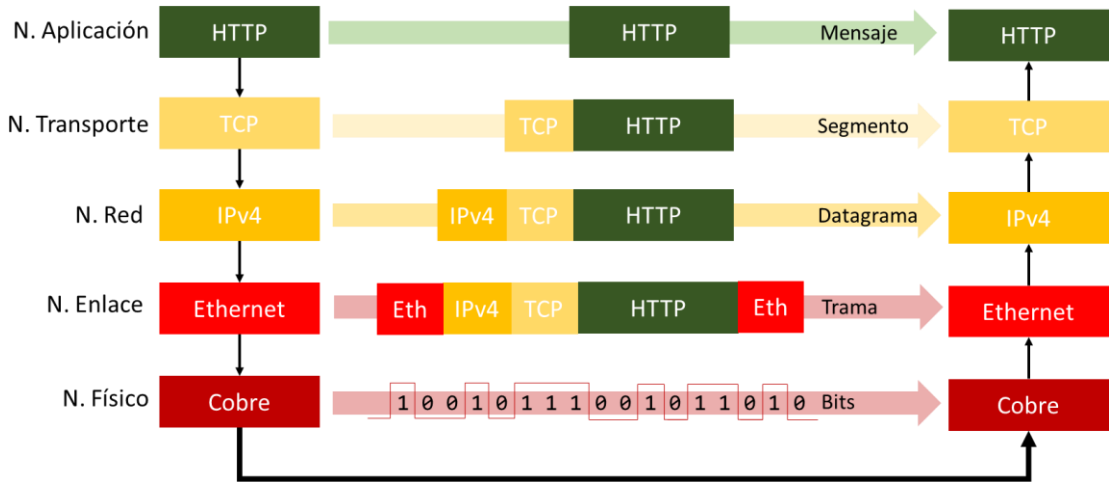



**Figura 2.3** – Comunicaciones entre protocolos de la torre TCP/IP

La capa Ethernet procesará su cabecera, verá que está destinada a ese equipo y que contiene datos del protocolo IPv4 (las cabeceras de todos los protocolos suelen incluir un campo de multiplexación, que indica a qué protocolo de nivel superior pertenecen los datos que transporta) y se lo entregará a IPv4, después de quitar su cabecera. Así que la capa IPv4 del servidor web recibirá exactamente el mismo datagrama que envió el navegador web. IPv4 analizará la cabecera y si es correcta entregará los datos que contiene (eliminando su cabecera) a TCP. Por lo tanto, TCP también recibirá exactamente el mismo segmento que envió el protocolo TCP del navegador web, y repetirá la operación, después de analizar su cabecera, y finalmente le entregará el mensaje HTTP al servidor web. Si el servidor web quiere enviarle una respuesta al navegador web, se repetirán exactamente las mismas operaciones, pero en sentido contrario.

Un aspecto interesante es que, si alguien es capaz de recibir e interpretar la señal electromagnética que atraviesa el cable entre los dos equipos, puede reconstruir la comunicación completa entre ambos equipos, puesto que los datos enviados incluyen las cabeceras Ethernet + cabecera IP + cabecera TCP + mensaje HTTP. Eso es justamente lo que hacen los analizadores de protocolos (e.g. 'tcpdump' o 'Wireshark'): capturar el tráfico que se envía por un enlace o una red y analizar todos los protocolos que incluyen los paquetes.

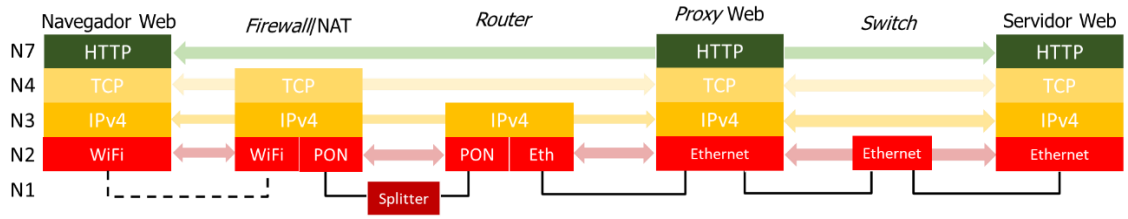



**Figura 2.4** – Elementos de interconexión en redes TCP/IP

Pero, obviamente, en Internet los ordenadores no están conectados directamente entre ellos, sino que utilizan una serie de nodos de interconexión, que son capaces de funcionar en diferentes niveles de la torre TCP/IP (**Figura 2.4**). Lo más importantes son:

- ***Splitters* y *Hubs***: Son elementos de interconexión a nivel físico (N1), y se encargan de amplificar y reenviar la misma señal que reciben por el resto de las fibras ópticas o puertos de cobre que tienen (respectivamente).
- ***Switches* Ethernet y puntos de acceso WiFi:** Son elementos de interconexión a nivel de enlace (N2) que son capaces de identificar a que estación o estaciones de la red de área local van destinadas las tramas que reciben y reenviárselas.
- ***Router* IP:** Son elementos de interconexión a nivel de red (N3) que son capaces de encaminar los datagramas IP que reciben a otros *routers* para ir avanzando a través de Internet hasta llegar al destino final de la comunicación.
- ***Firewalls* y *NATs*:** Son *routers* IP, pero que son capaces de interpretar (e incluso modificar) las cabeceras de nivel de transporte, para evitar que pase tráfico no permitido por las políticas de seguridad de la organización, o de traducir direcciones IP privadas en direcciones IP públicas (y viceversa), para que un equipo con direccionamiento privado pueda comunicarse con otros equipos en la Internet pública.
- ***Proxies* Web y Servidores de correo:** Son elementos de interconexión de nivel de aplicación (N7) que son capaces de interpretar los mensajes de nivel de aplicación, por ejemplo, para cachear contenidos web populares o hacer que un correo electrónico llegue hasta sus destinatarios.

Una vez que hemos repasado el funcionamiento de la torre de protocolos TCP/IP y la terminología que emplea, vamos a repasar los principales protocolos que forman parte de la misma.

## 2.2. Ethernet/IEEE 802.3

IEEE 802.3 o Ethernet es el protocolo de nivel de enlace para redes de área local (LAN) más usado globalmente, debido a su alta velocidad (que va de 10 Mbps a 400

Gbps) y bajo coste. De hecho, el resto de protocolos de enlace multipunto como IEEE 802.11/WiFi suelen interoperar nativamente con él, e incluso pueden emular que son un interfaz Ethernet (e.g. los chips WiFi pueden entregar los datos que reciben como tramas Ethernet), así que la mayoría de las capturas de tráfico incluyen una cabecera Ethernet, incluso si el interfaz de captura no lo es.

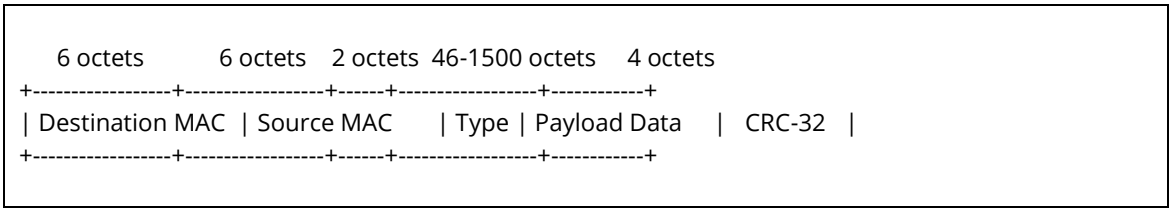



**Figura 2.5** – Formato de una trama Ethernet

La **Figura 2.5** muestra el formato de una trama Ethernet (ignorando los campos de Preámbulo de 8+1 octetos e *Inter-Frame Gap* de 12 octetos que se utilizan para delimitar la trama a nivel físico, pero que nunca aparecen en las capturas de tráfico).

Las estaciones conectadas a una red de área local IEEE (eso incluye tanto a Ethernet/802.3 como a WiFi/802.11 y Bluetooth/802.15) se identifican mediante direcciones MAC (*Medium Access Control*), que ocupan 48 bits (6 octetos), vienen grabadas en el propio interfaz de red (aunque se pueden cambiar vía software), y normalmente son únicas globalmente (i.e. no debería haber dos direcciones MAC repetidas en todo el mundo). Esto es posible porque las direcciones MAC tienen dos partes:

- **OUI** (*Organizationally Unique Identifier*): Son los primeros 3 octetos de la dirección MAC e identifican al fabricante del interfaz de red (e.g. el OUI e4:b9:7a está asignado a Dell). Así que puede ser muy útil para identificar al fabricante de un dispositivo (o al menos al del interfaz de red). Los últimos dos bits del primer octeto del OUI indican si la dirección MAC es globalmente única (0) o no (1), y si se trata de una dirección *unicast* (0) o *multicast* (1). Por ejemplo la dirección de *broadcast* ff:ff:ff:ff:ff:ff tiene ambos bits a 1.
- **NIC** (*Network Interface Controller*): Son los 3 octetos restantes de la dirección MAC, y cada fabricante encarga que cada interfaz de red que fabrique tenga una NIC diferente (i.e. podría considerarse como el número de serie de la tarjeta de red).

Así que los campos MAC Destino (*Destination* MAC) y MAC Origen (*Source* MAC) indican la estación (o estaciones en el caso de direcciones *broadcast*/*multicast*) que debe recibir la trama, y la estación que la envió, respectivamente.

Normalmente las tarjetas de red Ethernet filtran todo el tráfico que reciben que no vaya a su dirección MAC o las direcciones de *broadcast* y *multicast* a la que estén suscritas. Si se quiere cambiar este comportamiento para que una tarjeta de red

Ethernet pueda capturar todo el tráfico que reciba, hay que ponerla en **modo promiscuo** (*promiscuous mode*), para lo que típicamente se necesitan permisos de superusuario.

El campo Tipo Ethernet (*EtherType*) indica cual es el protocolo de nivel superior al que pertenecen los datos. Por ejemplo, IPv4 utiliza el Tipo=0x0800, IPv6 el Tipo=0x86DD y ARP el Tipo=0x0806.

Por motivos históricos las tramas Ethernet deben tener un tamaño mínimo de 64 octetos, por lo que los datos deben ocupar al menos 46 octetos (y de lo contrario se rellenan con ceros), y tienen un tamaño máximo o MTU (*Maximum Transmission Unit*) de 1500 octetos, aunque hay redes Ethernet que permiten tramas de hasta 9000 octetos (*jumbo frames*).

Por último, el campo *Frame Check Sequence* (FCS) es un *checksum* de toda la cabecera Ethernet y los datos que transporta, utilizando el algoritmo CRC-32, que es mucho más robusto que los *checksum* de los protocolos TCP/IP. Cuando se detecta un fallo en el *checksum* de una trama Ethernet, la tarjeta de red la descarta y no se entrega al sistema. De hecho, como tanto la generación como la comprobación del *checksum* lo realiza el hardware de la tarjeta de red, normalmente no aparece cuando se captura tráfico, con lo normalmente en los ficheros de captura solo aparece la cabecera inicial de la trama Ethernet (14 octetos).

Curiosamente, aunque la gran mayoría de las tramas Ethernet utilizan este formato, el formato oficial definido por el IEEE en el estándar 802.3 es ligeramente diferente, porque el campo Tipo se reemplaza por la Longitud de los datos (se puede diferenciar entre ellos porque todos los Tipos Ethernet tienen un valor superior a 1500), e incluye obligatoriamente una cabecera LLC (*Local Link Control*) y a veces otra SNAP (*Subnetwork Access Protocol*) para identificar el protocolo al que pertenecen los datos. Aunque este formato de trama alternativo solo lo utilizan otros protocolos IEEE, como por ejemplo STP (*Spanning Tree Protocol*).

Lo que sí es más habitual, sobre todo en redes empresariales, es que aparezca un campo de 4 octetos entre la dirección MAC origen y el campo Tipo, que contiene el identificador de la VLAN (red de área local virtual) a la que pertenece dicha trama. Ese campo opcional se puede añadir porque sus dos primeros octetos tienen el valor fijo 0x8100 como si fueran un Tipo Ethernet, seguido del identificador de VLAN de 2 octetos y de nuevo otro campo Tipo de 2 octetos, que esta vez sí indica el protocolo al que pertenecen los datos.

En las redes de área local (LAN) modernas los equipos finales se conectan a unos equipos de comunicaciones denominados conmutadores o *switches* Ethernet, que permiten la comunicación *full-duplex* (i.e. permiten enviar y recibir tramas simultáneamente) a alta velocidad (e.g. 1-40Gbps), y típicamente sin necesidad de

protocolos adicionales para asignar direcciones (porque cada tarjeta de red viene de fábrica con su propia MAC), ni de encaminamiento (aunque la redes con enlaces redundantes requieren el protocolo STP para garantizar que no hay bucles en la red), por lo que Ethernet es una tecnología casi *plug-and-play*. Esto es posible porque los *switches* Ethernet son capaces de aprender dónde están todas las estaciones de la LAN, simplemente recordando las direcciones MAC origen de todas las tramas que procesa y el puerto por el que le llegan. Si no sabe cómo llegar a una dirección MAC nueva, o alguna estación envía una trama a una dirección de *multicast*/*broadcast*, el *switch* simplemente envía la trama por todos sus puertos, excepto por el que le llegó. Una vez que aprende dónde está cada dirección MAC de la LAN, enviará todas las tramas destinadas a cada una de ellas únicamente por el puerto donde ésta se encuentre. Por lo que, si conectamos una estación de captura de tráfico a un *switch* Ethernet, incluso activando el modo promiscuo, típicamente solo podrá captura las tramas dirigidas a su dirección MAC, el tráfico *broadcast*/*multicast* de la LAN, y ocasionalmente alguna dirección MAC que el *switch* todavía no conozca. En el siguiente capítulo veremos formas de capturar tráfico en redes Ethernet.

## 2.3. Internet Protocol (IP)

IP (*Internet Protocol*) es, literalmente, el protocolo más importante de Internet. De hecho, Internet se podría definir como el conjunto de equipos que implementan el protocolo IP. El protocolo IP es un protocolo de nivel de red que se encargar de reenviar los datagramas IP entre dos puntos cualesquiera de Internet, aunque no garantiza su entrega de manera fiable (i.e. los datagramas se pueden perder, desordenar, e incluso duplicar).

Actualmente existen dos versiones del protocolo IP: IPv4 [RFC791], que utiliza direcciones de 32 bits, e IPv6 [RFC8200] que utiliza direcciones de 128 bits. Aunque ya no quedan direcciones IPv4 públicas disponibles en Internet (que es la principal razón por la que se diseñó IPv6), el uso de direcciones privadas y NATs (*Network Address Translation*) para interconectarlas a la Internet pública, hace que el uso de IPv4 seguirá siendo mayoritario durante bastantes años.

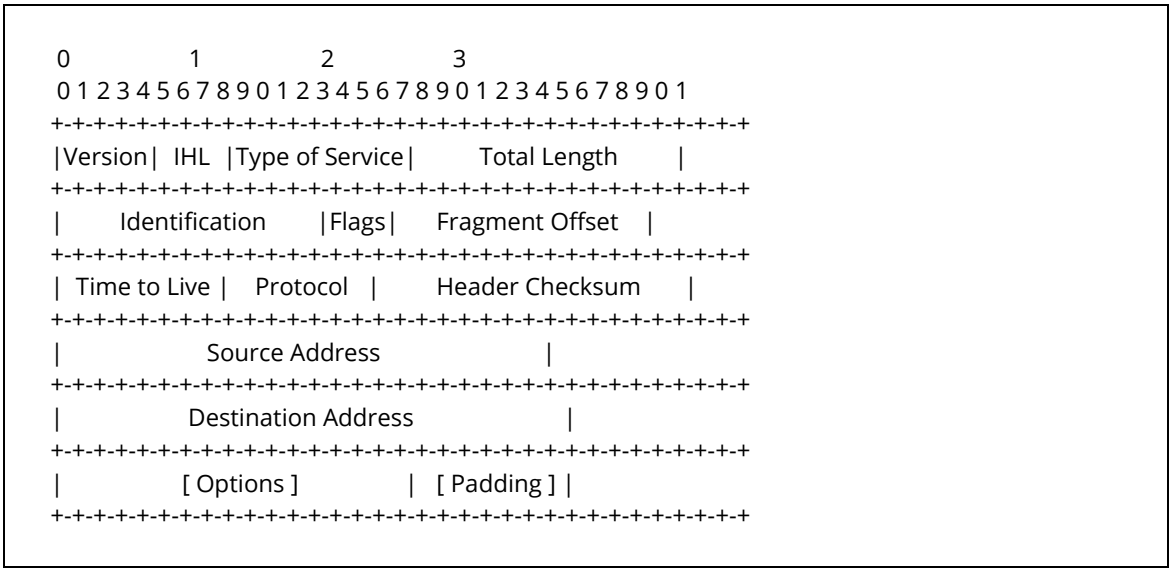



**Figura 2.6** – Formato de la cabecera IPv4 [RFC791]

La **Figura 2.6** muestra el formato de la cabecera IPv4. Se denomina así porque los primeros 4 bits de la misma contienen el valor 0x4. Aunque se han definido más versiones del protocolo IP, las únicas que se han utilizado comercialmente son IPv4 e IPv6.

El campo IHL (*Internet Header Length*) indica la longitud de la cabecera IPv4 en palabras de 32 bits, por lo que un valor 0x5 indica que la cabecera IPv4 ocupa 20 octetos, y por los tanto no tiene opciones (que es lo más habitual, puesto que actualmente todas están en desuso).

El campo Tipo de Servicio (*Type of Service* - ToS) se definió para indicar la importancia de cada datagrama y así proporcionar calidad de servicio (*Quality of Service* - QoS) en Internet. Aunque su sintaxis se ha redefinido varias veces (e.g. Diffserv), no se suele utilizar y normalmente todas las aplicaciones lo dejan a 0x00.

El campo *Total Length* indica el tamaño en octetos del datagrama IPv4 completo, incluyendo la cabecera y los datos, por lo que el tamaño máximo de un datagrama IPv4 está limitado a 65,535 octetos. Aunque normalmente son mucho menores, porque para poder transportarse sobre niveles de enlace como Ethernet (que tienen una MTU de 1500 octetos), es necesario que los *routers* los dividan en múltiples fragmentos y el sistema final los reensamble, con la pega de que la pérdida de cualquiera de los fragmentos hace que se pierda el datagrama IPv4 completo.

Los campos *Identification*, *Flags* y *Fragment Offset* se utilizan justamente para identificar a los fragmentos de un mismo datagrama, marcar si el datagrama debe fragmentarse o no (y en ese caso si quedan más fragmentos pendientes o es el último), e indicar qué parte de los datos contiene este fragmento. Sin embargo, en

la Internet actual no es habitual el uso de fragmentación IPv4, porque el protocolo de transporte TCP implementan su propia segmentación de datos (aunque UDP no). Además, como la cabecera del protocolo de transporte solo aparece en el primer fragmento, hay muchos *firewalls* que descartan el resto de los fragmentos.

El campo *Time To Live* (TTL) indica el número de veces que puede reenviarse este datagrama, antes de descartarse y enviar un mensaje ICMP de error al origen. Este mecanismo sirve para detectar y mitigar bucles en el encaminamiento dentro de la red (e.g. si dos *routers* se tienen como siguiente salto hacia un destino, los paquetes con ese destino se reenviarían de uno a otro hasta que se agote el TTL). Los diferentes sistemas operativos utilizan diferentes valores de TTL para los paquetes que generan. Los UNIX (incluido Linux y macOS) suele utilizar un TTL=64, mientras que Windows utiliza un TTL=128, aunque en muchos de ellos los mensajes ICMP suelen emplear el valor máximo TTL=255.

El campo *Protocol* indica cual es el protocolo de nivel superior al que pertenecen los datos que transporta este datagrama. Por ejemplo, TCP tiene asignado el protocolo 6, UDP el 17, ICMP el 1, e IPv6 el 41 (e.g. para crear túneles IPv6 sobre IPv4).

El campo *checksum* sólo cubre la propia cabecera IPv4 (los protocolos de transporte como TCP y UDP tienen su propio campo *checksum*) y es bastante débil desde el punto de vista de detección de errores (comparado con otros algoritmos de detección de errores como CRC-32 de que usa Ethernet), puesto que no es más que la suma en complemento a 1 de todos los octetos de la cabecera cogidos de dos en dos. Esto hace que la probabilidad de que un datagrama IPv4 corrupto tenga un *checksum* correcto no sea despreciable.

Las direcciones IP origen y destino indican cuál es el equipo que envió el datagrama y el destino del mismo. Así que todos los equipos conectados a Internet deberían tener una **dirección IPv4 pública** (i.e. única globalmente). Sin embargo, esto no ocurre en la actualidad, puesto que la mayoría de los equipos utiliza una **dirección IPv4 privada**, de alguno de estos rangos: 10.0.0.0/8, 172.16.0.0/12, 192.168.0.0/16, que utilizan para comunicarse dentro de su organización. Sin embargo, para comunicarse con equipos en Internet, es necesario un *router* intermedio con capacidades de NAT (*Network Address Translation*) que cambia la dirección IP origen privada por una dirección IP pública (normalmente también se modifica el puerto origen, para que la misma dirección IP pública pueda utilizarse para varias direcciones IP privadas), y realice en cambio inverso sobre las direcciones IP destino en los paquetes que vuelven de Internet. Así que, dependiendo del lugar donde se realiza la captura de tráfico, pueden verse únicamente direcciones IP privadas (comunicaciones dentro de una organización), solo direcciones IP públicas (si se captura después del NAT de salida de la organización o en servidores con

direcciones IP públicas en la DMZ de la organización), o direcciones IP privadas y públicas (en comunicaciones con el exterior de la organización).

A diferencia de las direcciones MAC, los interfaces de red IPv4 no tienen direcciones IP preasignadas, por lo que hay que asignar una (o más) direcciones IP a cada interfaz de red que se quiera usar para enviar y recibir tráfico. En las estaciones de captura se recomienda que los interfaces de red utilizados para capturar tráfico no tengan ninguna dirección IP configurada, para que así no puedan comunicarse con ella. Las direcciones IP se pueden configurar de manera estática (típicamente para servidores), de manera dinámica con algún protocolo como DHCP o PPP. En cualquier caso, es muy importante garantizar que dentro de la misma subred no haya dos equipos con la misma dirección IP configurada.

Otra diferencia con Ethernet es que en IP ya no es viable aprender cómo llegar a cada una de las 2^32 direcciones IPv4 que existen en Internet (vs. los cientos o miles de direcciones MAC que puede haber en una LAN), por lo que las direcciones IP se agrupan en **subredes**, que agrupan a todas las direcciones IP que comparten el mismo prefijo. Por ejemplo, la subred 192.168.1.0/255.255.255.0 (o simplemente 192.168.1.0/24) está formada por todas las direcciones IP cuyos primeros 24 bits empiecen por 192.168.1.0, como por ejemplo 192.168.1.1, 192.168.1.33 o 192.168.1.110. Mientras que la dirección IP 192.168.2.100, no pertenece a esa subred.

**Listado 2.7** – Tabla de rutas de un equipo TCP/IP

```
$ ifconfig
eth0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
      inet 192.168.1.33  netmask 255.255.255.0  broadcast 192.168.1.255
      inet6 fe80::ec1c:80a4:5f2c:1426  prefixlen 64  scopeid 0x20<link>
      ether 00:15:5d:38:01:04  txqueuelen 1000  (Ethernet)
      RX packets 5465154  bytes 1071721836 (1.0 GB)
      RX errors 0  dropped 32269  overruns 0  frame 0
      TX packets 1936312  bytes 369934714 (369.9 MB)
      TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

lo: flags=73<UP,LOOPBACK,RUNNING>  mtu 65536
      inet 127.0.0.1  netmask 255.0.0.0
      inet6 ::1  prefixlen 128  scopeid 0x10<host>
      loop  txqueuelen 1000  (Bucle local)
      RX packets 1260904  bytes 3032676040 (3.0 GB)
      RX errors 0  dropped 0  overruns 0  frame 0
      TX packets 1260904  bytes 3032676040 (3.0 GB)
      TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

$ route -n
Tabla de rutas IP del núcleo
Destino        Pasarela       Genmask        Indic Métric Ref  Uso Interfaz
0.0.0.0        192.168.1.1    0.0.0.0        UG    100    0      0 eth0
192.168.1.0    0.0.0.0        255.255.255.0  U     100    0      0 eth0
```

Además de una dirección IP, todos los equipos que implementen el protocolo IP deben tener una **tabla de rutas** que le indique al equipo como llegar a una dirección IP destino dada. El **Listado 2.7** muestra los interfaces de red y la tabla de rutas de un equipo TCP/IP con sistema operativo Linux. En particular, el equipo tiene dos interfaces de red: el interfaz "eth0", que es un interfaz Ethernet con dirección MAC 00:15:5d:38:01:04 y que se ha configurado con la dirección IP privada 192.168.1.33, y el interfaz "lo" configurado con la dirección IP 127.0.0.1, que es un interfaz especial denominado ***loopback*** y que tienen todos los equipos IP para que los procesos dentro de la máquina puedan comunicarse utilizando IP (y que no puede comunicarse con el exterior del sistema).

La tabla de rutas del equipo es muy simple, solo tiene dos entradas: Una que indica la subred a la que pertenece el interfaz "eth0": 192.168.1.0/255.255.255.0, de forma que pueda comunicarse directamente con cualquier equipo de la misma subred, sin necesidad de pasar por un router o *gateway*. La segunda entrada se denomina **ruta por defecto**, porque indica que para ir a cualquier otra dirección IP de Internet (0.0.0.0/0.0.0.0 o simplemente 0.0.0.0/0) se debe reenviar los datagramas al *router* por defecto 192.168.1.1.

Sin entrar en muchos detalles, la ruta para ir a un destino se elige utilizando el algoritmo de ***Longest Prefix Match*** que, en caso de que existan múltiples rutas posibles hacia el mismo destino, elige aquella que tenga el prefijo común con la dirección IP destino más largo. Por ejemplo, para comunicarse con la dirección IP destino 192.168.1.110, se podría utilizar tanto la primera ruta (192168.1.0/24) como la segunda (0.0.0.0/0), porque la ruta por defecto cubre cualquier dirección IPv4 del mundo. Sin embargo, se elige la primera, ya que la dirección IP tiene 24 bits en común con la primera ruta y ningún bit en común con la segunda, por lo que el datagrama se puede enviar directamente al destino, si pasar por ningún *router*. Por el contrario, si se quisiese ir a la dirección IPv4 93.184.216.34, (que no pertenece a la subred 192.168.1.0/24), la única ruta posible es la por defecto (0.0.0.0/0), y el datagrama se reenviaría al *router* con dirección IP 192.168.1.1 (el datagrama mantendría la dirección IP destino original, pero se enviaría en una trama Ethernet a la dirección MAC del *router*).

¿Y qué haría el *router*? Un *router* IP no es más que un equipo TCP/IP con múltiples interfaces de red y que está configurado para que pueda reenviar datagramas IP entre ellas. Esto es, un equipo que puede recibir datagramas IP no destinados a él y reenviarlos por otro interfaz al siguiente salto hacia el destino (por lo que el siguiente salto puede ser otro *router* o el destino final). Los *routers* también utilizan una tabla de rutas para saber cómo llegar a los diferentes destinos, aunque normalmente tiene muchas más rutas que los equipos finales, porque utiliza un

protocolo de encaminamiento como RIP, OSPF, ISIS o BGP para aprender donde están las diferentes redes de la organización y/o de Internet.

Pero esta explicación tiene un problema. Si las tablas de rutas de IP sólo indican la dirección IP del siguiente salto al que enviar el datagrama (i.e. otro *router* o el destino final), pero las tarjetas de red Ethernet solo saben de direcciones MAC, ¿cómo puede saber el *router* la dirección MAC del siguiente salto? Para eso se utiliza el protocolo ARP.

## 2.4. Address Resolution Protocol (ARP)

El protocolo de resolución de direcciones (ARP) [RFC826] permite obtener, en enlaces multipunto (como las redes Ethernet), la dirección de enlace asociada a una dirección IPv4 dada.

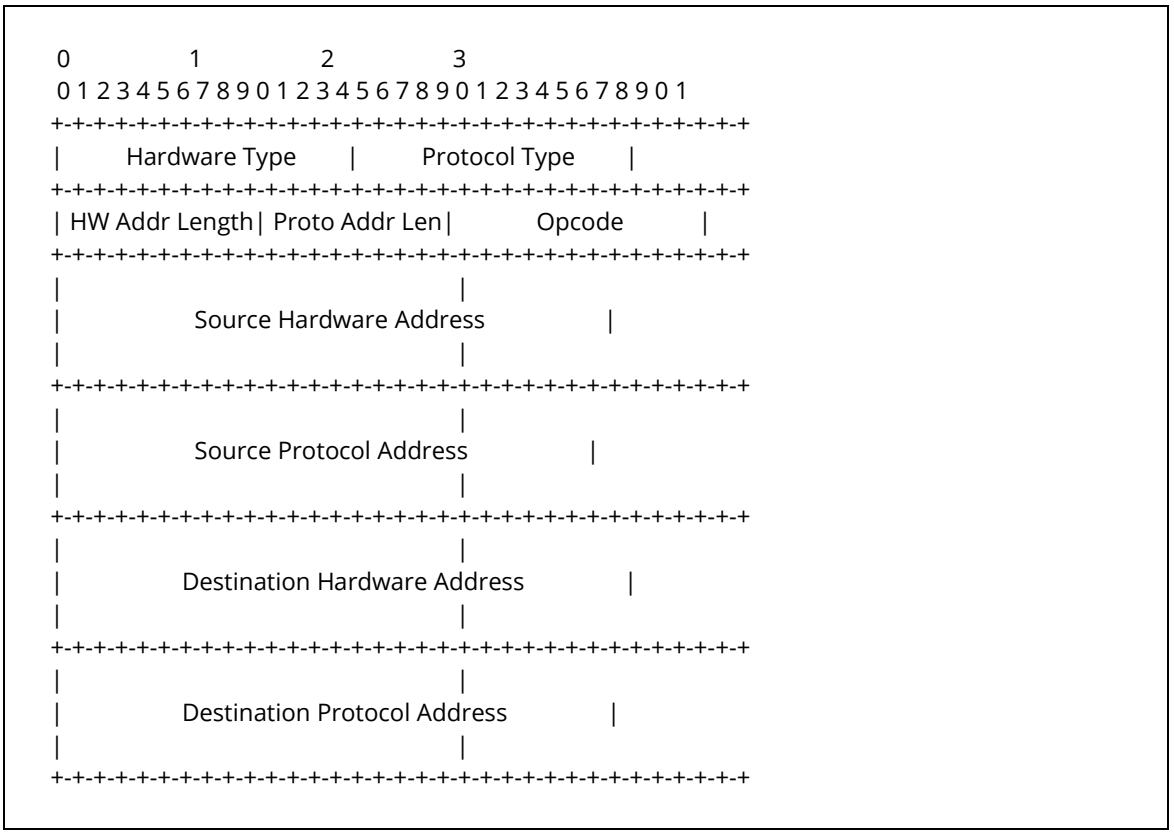



**Figura 2.8** – Formato de los mensajes ARP

Aunque ARP se diseñó para soportar cualquier tipo de protocolo de enlace y de red (por eso permite indicar el tipo y longitud de las direcciones de enlace y red), habitualmente solo se emplea con Ethernet (HW Type=0x0001; *HW Addr. Length*=6) e IPv4 (*Proto Type*=0x0800; *Proto. Addr. Length*=4).

El protocolo ARP es muy sencillo. Si algún equipo de la LAN quiere saber cuál es la dirección MAC asociada a una dirección IPv4, simplemente manda en una trama

Ethernet *broadcast* (i.e. con dirección MAC destino ff:ff:ff:ff:ff:ff) un mensaje ARP (*EtherType*=0x0806) de petición de resolución (*Opcode*=1), que contiene la dirección IP y MAC del origen (*Sender HW & Protocol Address*), y la dirección IP por la que se pregunta (*Destination Protocol Address*). Todos los equipos de la LAN recibirán el mensaje ARP y lo procesan para ver si pregunta por su dirección IP. Si no lo hace, simplemente ignoran el mensaje y no envían ninguna respuesta. Pero si un equipo tiene la dirección IP por la que se pregunta, responderá con un mensaje ARP de respuesta (*Opcode*=2) con sus direcciones IP y MAC (*Source HW & Protocol Address*), y que se envía directamente a la dirección MAC del origen, para no molestar al resto de equipos de la subred.

Aunque el protocolo es muy sencillo, enviar una trama Ethernet a todos los equipos de la subred y esperar respuesta cada vez que se quiere enviar un datagrama IP no sería muy eficiente, por lo que todas implementaciones TCP/IP tienen una **cache ARP**, que almacena durante cierto tiempo la dirección MAC asociada a cada dirección IP obtenida a través de ARP (ver **Listado 2.9**). De hecho, para evitar que el destino tenga que preguntar por nuestra dirección MAC cuando nos envíe respuesta, en la cache ARP también se introducen las direcciones IP/MAC origen de las peticiones. Otra optimización para evitar tráfico *broadcast* consiste en actualizar las entradas expiradas de la cache ARP preguntando directamente a la dirección MAC almacenada. De forma que cuando un equipo cambia su dirección MAC (que no es una operación frecuente, pero puede ocurrir con protocolos de alta disponibilidad como VRRP), también puede anunciar su nueva MAC a toda la subred de manera proactiva, enviando un mensaje ARP de respuesta no solicitado (*unsolicited* ARP, en inglés). Esta "promiscuidad" de ARP a la hora de aprender direcciones MAC da pie a ataques de envenenamiento de cache ARP, que por otro lado nos pueden permitir capturar tráfico de una estación en la misma subred, como veremos en el próximo capítulo.

**Listado 2.9** – Ejemplo de cache ARP

```
$ arp -n
Dirección          TipoHW  DirecciónHW         Indic Máscara     Interfaz
192.168.1.1        ether   78:29:ed:dc:62:99   C                 eth0
192.168.1.110      ether   e4:b9:7a:b4:43:7e   C                 eth0
```

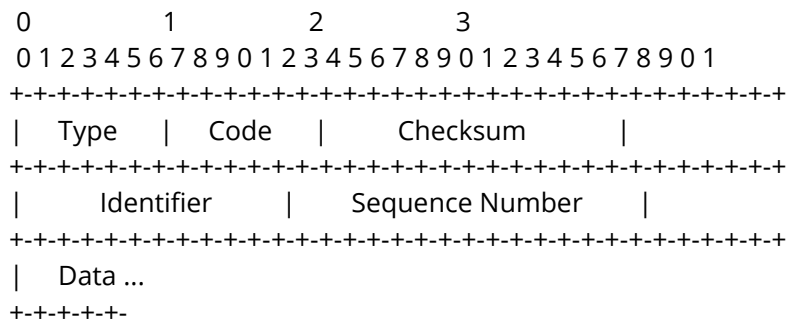
## 2.5. Internet Control Message Protocol (ICMP)

El *Internet Control Message Protocol* (ICMP) [RFC792] es el protocolo de control de la pila de protocolos TCP/IP, y se utiliza, además de para comprobar la conectividad IP entre equipos conectados a Internet (e.g. 'ping', 'traceroute'), para señalizar errores, como cuando no se puede alcanzar un destino, porque no existe una ruta para

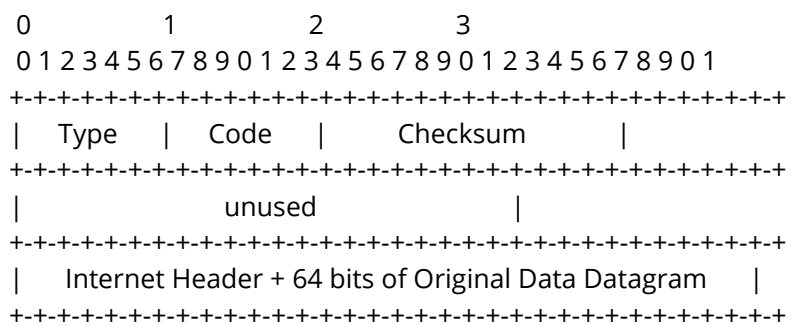
llegar hasta él o se utiliza un protocolo o un puerto UDP que no está en uso. ICMP se envía directamente sobre datagramas IPv4 (*IPv4.proto* = *1*). IPv6 tiene su propio protocolo ICMPv6 [RFC4443] que, aunque tiene un uso y una estructura similar al protocolo ICMP de IPv4, también se utiliza para realizar el descubrimiento de vecinos [RFC4861] de IPv6, lo que reemplaza completamente al protocolo ARP y parcialmente a DHCP.

Aunque cada tipo de mensaje ICMP define su propia cabecera, todas tienen una estructura similar, y en particular deben comenzar por tres campos comunes a todas ellas: Tipo (*Type*), Código (*Code*) y *Checksum*, que permiten identificar la estructura del resto de la cabecera ICMP. En los mensajes de error se suelen incluir la cabecera IPv4 y al menos los primeros 64 bits (i.e. para incluir los puertos de la cabecera TCP o UDP) del datagrama que ha generado el error para el origen sepa cuál de sus datagramas ha dado problemas.

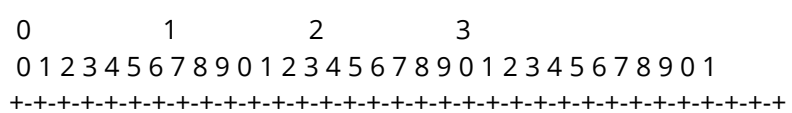
**Echo** (Type = 8) / **Echo Reply** (Type = 0) **Message**




**Destination Unreachable** (Type = 3) / **Time Exceeded** (Type = 11) Message




**Redirect Message** (Type = 5)



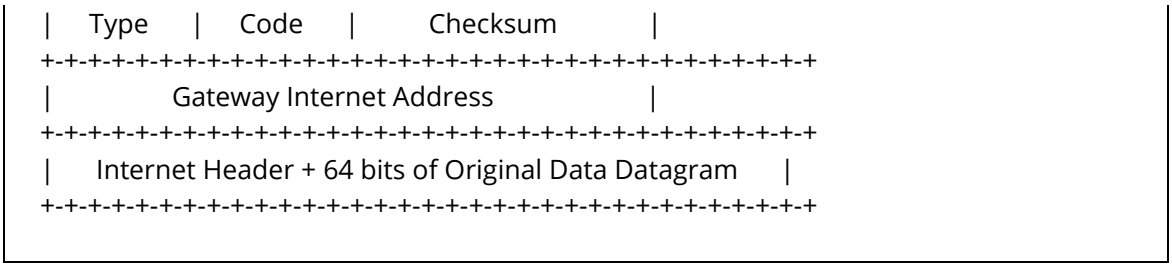



**Figura 2.10** – Formatos de los mensajes ICMP más importantes [RFC792]

La **Figura 2.10** muestra la estructura de los principales mensajes ICMP:

- ***Echo*** (Tipo ICMP = 8) / ***Echo Reply*** (Tipo ICMP = 0): Esta pareja de mensajes se utiliza para comprobar la conectividad IP entre dos equipos conectados de red, y son los que utiliza el comando 'ping'. El emisor envía un mensaje ICMP *Echo* al destino, y éste le responde con un mensaje ICMP *Echo Reply*, copiando todos los campos del mismo (i.e. simplemente tiene que cambiar el Tipo = 0).
- ***Time Exceeded*** (Tipo ICMP = 11): Este mensaje se genera cuando un *router* tiene que reenviar un datagrama IP con un tiempo de vida (TTL) igual a cero, por lo que debe descartarlo y mandar un mensaje ICMP de este tipo a la dirección IP origen para notificárselo. Este tipo de mensajes ICMP suele aparecer cuando existe algún tipo de bucle en la red, y por tanto nunca se llega al destino antes de que el TTL del datagrama se vaya decrementando hasta cero. Aunque también es el mensaje en el que se basa el comando 'traceroute' para tratar de identificar los *routers* intermedios que hay en la ruta hasta un destino. 'traceroute' envía datagramas IPv4 (mensajes ICMP *Echo* o datagramas UDP a un puerto efímero) al destino. Primero con un TTL=1, luego con TTL=2, etc. y así recibe los mensajes ICMP de todos los *routers* intermedios, hasta llegar al destino (i.e. que devuelve un mensaje ICMP *Echo Reply*, o un mensaje ICMP *Destination Unreacheable - Port Unreacheable* en caso de enviar datagramas UDP).
- ***Destination Unreacheable*** (Tipo ICMP = 3): Este mensaje se genera cuando el destino no puede alcanzarse por algún motivo. Lo más importantes serían:
    - ***Net Unreacheable*** (Código = 0): El *router* no tiene ninguna ruta válida para llegar a la dirección IPv4 destino, por ejemplo, si hay un problema de encaminamiento en la red.
    - ***Host Unreacheable*** (Código = 1): No es posible alcanzar el equipo destino, por ejemplo, si éste no responde a las peticiones ARP
    - ***Protocol Unreacheable*** (Código = 3): El datagrama ha llegado al equipo destino, pero éste no implementa el protocolo que transporta IPv4 (e.g. SCTP).
    - ***Port Unreacheable*** (Código = 4): El datagrama ha llegado al equipo destino y es de un protocolo conocido, pero el puerto destino no está

en uso. Normalmente este mensaje solo se utiliza para UDP, porque TCP puede señalizar este error.

- ***Redirect*** (Tipo ICMP = 5): Este mensaje se genera cuando un *router* detecta que el origen tiene un problema de encaminamiento, típicamente porque el *router* tiene que volver a reenviar el paquete por el mismo interfaz por el que lo recibió. Además de incluir la cabecera del datagrama problemático, se puede indicar la dirección de otro router al que deberían enviarse los datagramas hacia ese destino. Como los mensajes ICMP no están autenticados y se pueden falsificar, no es recomendable procesar este mensaje porque permite a un atacante cambiar la tabla de rutas de manera trivial.

## 2.6. User Datagram Protocol (UDP)

El *User Datagram Protocol* (UDP) [RFC768] es probablemente el protocolo de transporte más sencillo posible. Tal como muestra la **Figura 2.11**, su cabecera solo tiene 4 campos de 16 bits: puerto origen, puerto destino, longitud del datagrama UDP (incluyendo cabecera y los datos que transporta), y un *checksum* para comprobar que no se ha corrompido durante su transmisión.

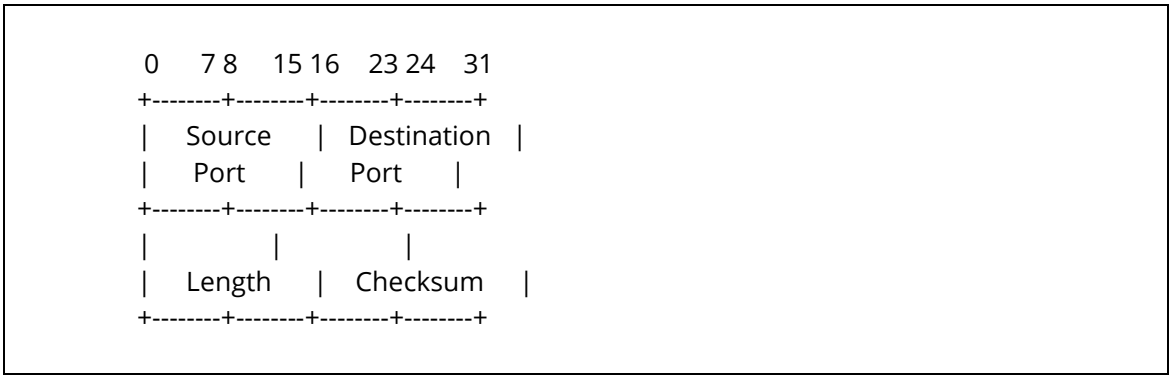



**Figura 2.11** – Formato de la cabecera del protocolo UDP [RFC768]

Los campos más importantes de la cabecera UDP son los puertos origen y destino que, junto a las direcciones IP origen y destino, definen las **direcciones de nivel de transporte** del origen (<dirección IP origen, protocolo UDP, puerto UDP origen>) y del destino (<dirección IP destino, protocolo UDP, puerto UDP destino>), lo que permite identificar "unívocamente" (obviando la presencia de NATs) los procesos extremos de la comunicación. La dirección IP identifica el ordenador, y el puerto un proceso que corre en el mismo. La combinación de las direcciones de transporte origen y destino da lugar a la quíntupla: <dirección IP origen, puerto origen, protocolo de transporte, IP destino, puerto destino>, que permite identificar un **flujo** de paquetes en Internet (tanto en UDP como en TCP, o cualquier otro

protocolo de transporte). Al ser campos de 16 bits, cada equipo solo puede tener $2^{16}$ = 65.536 puertos diferentes para cada protocolo de transporte, aunque el puerto 0 está reservado, por lo que solo se pueden utilizar los puertos entre el 1 y el 65.535. Los puertos UDP y TCP están organizados en diferentes grupos:

- **Puertos bien conocidos (*well-known*, en inglés): 1-1023**. Estos puertos solo los deberían usar procesos del sistema, por lo que, en muchos sistemas operativos como los UNIX, es necesario tener permisos de superusuario para utilizar uno de estos puertos. La mayoría de los primeros protocolos que se definieron en Internet usan puertos en este rango, como por ejemplo: SSH (TCP/22), SMTP (TCP/25), DNS (UDP/53, TCP/53), HTTP (TCP/80), HTTPS (TCP/443). Normalmente estos puertos siempre se usan en el lado servidor (que debe tener una dirección de transporte conocida para el cliente), aunque hay excepciones como BOOTP/DHCP, donde el servidor utiliza el puerto UDP/67, y el cliente el puerto UDP/68.
- **Puertos registrados: 1024-49151**. La popularidad de Internet y la explosión de protocolos hizo que el rango de los puertos bien conocidos se agotase rápidamente, por lo que el rango de puertos reservados se amplió hasta este grupo. Estos puertos también suelen estar reservados para la parte servidora de los protocolos, aunque normalmente no es necesario tener permisos de superusuario para utilizar uno de estos puertos.
- **Puertos dinámicos o efímeros: 49152–65535**. Estos puertos no pueden reservarse para ningún protocolo, porque los usan los clientes de los diferentes protocolos para comunicarse con los servidores. Normalmente es el sistema operativo el que selecciona un puerto al azar o el primero que se encuentre disponible, por lo que cada vez que se ejecuta un cliente puede utilizar un puerto origen diferente. Esto no es mayor problema en las aplicaciones cliente-servidor porque, como las comunicaciones las inicia el cliente y la dirección de transporte del servidor es bien conocida (i.e. utiliza un puerto fijo en el rango 1-1023 o 1024-49151), el servidor únicamente tiene que responder a la dirección de transporte origen del cliente. Así que, en general, es posible identificar si un datagrama UDP lo ha enviado un cliente o un servidor simplemente comprobando si el puerto origen o el destino pertenecen a este rango.

La lista oficial de puertos asignados [PortList] la mantiene el IANA (*Internet Assigned Numbers Authority*), aunque hay muchos protocolos que no registran su uso (especialmente en el caso de *malware* o protocolos propietarios), por lo que también es recomendable consultar páginas como [SpeedGuide], que van registrando los usos no oficiales de cada puerto.

El campo longitud de UDP indica el número de octetos que ocupa tanto la cabecera UDP como los datos que transporta. Por lo tanto, el mensaje UDP más largo solo

puede tener 65.536 – 8 = 65.628 octetos, aunque depende de IP para fragmentar mensajes que no quepan en un datagrama IP. Los mensajes UDP fragmentados suelen dar muchos problemas, ya que la pérdida de un único fragmento impide la recepción del mensaje completo. Además, dado que la cabecera UDP solo aparece en el primer fragmento, hay *firewalls* que filtran el resto de los fragmentos. Por esta razón hay muchas aplicaciones UDP que limitan el tamaño de sus mensajes. Por ejemplo, los mensajes DNS sobre UDP por defecto solo pueden tener una longitud de 512 octetos. En general UDP se utiliza para el envío de mensajes pequeños (vs. TCP que se emplea para la transmisión de grandes cantidades de datos).

Por último, el campo *checksum* de UDP es una suma de comprobación de la cabecera UDP, de los datos que transporta, y de algunos campos de la cabecera IP (dado que las direcciones de transporte también están formadas por las direcciones IP y el identificador de protocolo de IP), lo que sirve para comprobar si algunos de estos datos se han corrompido durante el reenvío del datagrama UDP hasta el destino. Sin embargo, el algoritmo de *checksum* definido para UDP (que es el mismo para IPv4 y TCP), es muy débil y es posible que llegue un datagrama UDP con datos corruptos que no sean detectados por el *checksum*. De hecho, UDP incluso permite enviar datagramas con el campo *checksum* a 0, que no se comprueba en la recepción. Esto puede ser útil para aplicaciones, como las multimedia, que tienen cierta tolerancia a errores.

Como puede observarse, en la cabecera UDP no hay ningún campo de control o de número de secuencia (como sí tiene TCP), así que UDP no permite establecer una conexión con estado entre los equipos origen y destino, ni dispone de la capacidad de detectar si se pierde algún datagrama, por lo que obviamente no puede solicitar su retransmisión. Por lo tanto, UDP es un protocolo no orientado a conexión, no fiable y sin control de flujo ni de congestión (en contraposición a TCP que es orientado a conexión, fiable y con control de congestión y de flujo). Así que es responsabilidad de la aplicación por encima de UDP de detectar si se pierden datos y retransmitirlos en caso necesario. Por otro lado, UDP permite enviar datos inmediatamente al destino (incluso si este es una dirección IP *broadcast* o *multicast*), y permite a la aplicación controlar exactamente la tasa y el momento en la que se envía su información, por lo que históricamente ha sido el protocolo más utilizado por las aplicaciones multimedia, así como por los ataques de denegación de servicio (DoS).

## 2.7. Transmission Control Protocol (TCP)

El *Transmission Control Protocol* (TCP) [RFC793] es el protocolo de transporte más importante de Internet, y por eso a la pila de protocolos de Internet se le conoce como TCP/IP. De hecho, el control de congestión de Internet (i.e. la razón por la que

las redes no se colapsan cuando se sobrecargan) se basa fundamentalmente en el comportamiento del protocolo TCP, que es el que transporta la inmensa mayoría de la información en Internet.

Tal como se puede observar en la **Figura 2.12**, TCP es un protocolo mucho más complejo que el protocolo UDP que analizamos en el apartado anterior, aunque mantiene algunas características similares como los dos campos de 16 bits para identificar los puertos origen y destino. O el campo *checksum*, que cubre tanto la cabecera TCP como los datos que transporta, así como las direcciones IP y el identificador de protocolo de la cabecera IP. Como estos campos ya se describieron en el capítulo sobre UDP, no se repetirán aquí.

Los campos más importantes de la cabecera TCP son los números de secuencia y de confirmación (de 32 bits cada uno), así como el conjunto de 6 *flags* (bits) de control, y el campo de ventana disponible (*Window*), que nos indican que nos encontramos ante un protocolo orientado a conexión, ordenado, fiable, y con control de flujo. Veremos ahora qué significan todos estos conceptos.

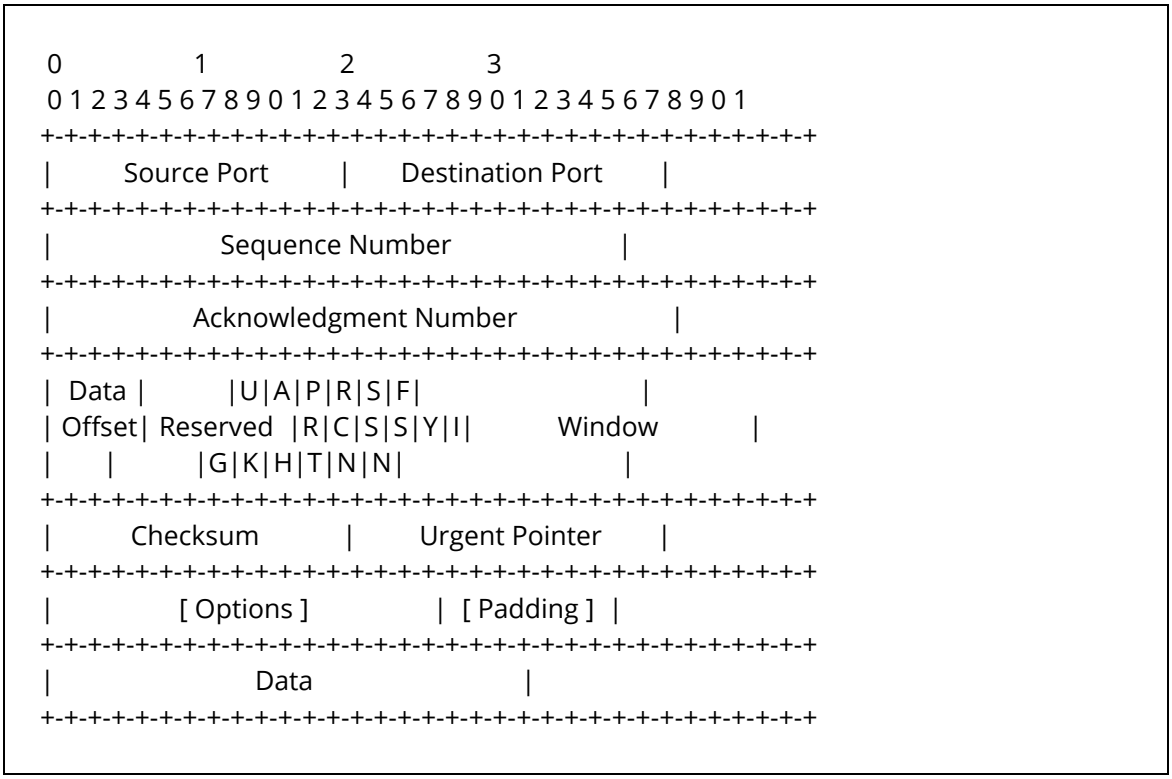



**Figura 2.12** – Formato de un segmento TCP [RFC793]

Todos los datos que envía TCP tienen un número de secuencia asociado (el campo *Sequence Number* es el número de secuencia del primer octeto de datos que transporta), cuya recepción correcta debe confirmar el receptor mediante el campo *Acknowledgment Number* (que realmente indica el siguiente número de secuencia que espera recibir). Por lo tanto, TCP es capaz de identificar si los segmentos llegan

desordenados (las redes IP no garantizan que los datagramas lleguen al destino o que lo hagan en el mismo orden que se enviaron) y reordenarlos (gracias a sus números de secuencia), así como detectar cuando se corrompe (aunque el algoritmo de *checksum* es el mismo que IPv4 y por lo tanto bastante débil) o pierde cualquier segmento de datos (al no recibir confirmación del mismo), y retransmitirlo automáticamente. De hecho, TCP ofrece un servicio de transmisión de datos ordenado, de forma que si tiene algún segmento pendiente no entregará ningún dato posterior a la aplicación. Algunas implementaciones de TCP permitían esto mediante el *flag* URG y el campo de Puntero Urgente, pero ya no se recomienda su uso. Así que en ese caso la recepción de datos se bloqueará hasta que se recupere el segmento perdido/desordenado. Esta es la razón por la que, hasta hace no mucho, todas las aplicaciones multimedia, en las que es más importante recibir datos de manera constante que recibir absolutamente todos los datos (porque es preferible perder un fotograma a congelar la reproducción de un vídeo), usaban UDP en lugar de TCP.

El **control de flujo** de TCP evita saturar al destino, enviando más datos de los que éste puede recibir. Para ello se utiliza el campo Ventana (*Window*), mediante el cual el receptor le indica al emisor el tamaño de *buffer* de recepción que tiene libre, de forma que el emisor nunca puede enviar más datos hasta que el receptor vuelva a "abrir" la ventana, indicando que tiene más *buffer* disponible.

Además de la ventana del receptor, el emisor debe tener una ventana de transmisión (que no tiene representación en ningún campo de la cabecera, al ser un estado interno), que limita la cantidad de información que se envía a la red, y que es la forma en la que se implementa el **control de congestión** en las redes TCP/IP. Este mecanismo es bastante complejo y, como es un factor fundamental en el rendimiento de TCP, han surgido bastantes variantes a lo largo de la historia, pero básicamente consiste en ir inyectando tráfico en la red poco a poco, cada vez más rápido hasta que se detecta que la red empieza a congestionarse (porque se detecta pérdida de paquetes o aumenta el tiempo de llegada de las confirmaciones), en cuyo caso se reduce drásticamente la tasa de envío, para volver a ir subiendo poco a poco e ir repitiendo el proceso continuamente para adaptarse a la capacidad disponible que tenga la red (idealmente todos los flujos TCP comparten equitativamente la capacidad disponible de la red).

Esta combinación de mecanismos hace que la aplicación que hace uso de TCP tenga un control muy limitado sobre cuándo o cómo se envían los datos (teóricamente el *flag PSH* indica a TCP que esos datos deberían entregarse inmediatamente, pero su efecto es limitado en la mayoría de las implementaciones de TCP), puesto que TCP puede retrasar su envío, dividir un mensaje en varios segmentos, esperar a que lleguen datos anteriores antes de entregarlos, etc. De hecho, TCP ofrece un servicio de envío de bytes de manera bidireccional, ordenada y fiable, pero no permite

agrupar bytes en mensajes (a diferencia de UDP), por lo que las aplicaciones sobre TCP deben definir esa semántica si la necesitan.

Todos estos mecanismos, y la necesidad de almacenar estado por cada flujo TCP en el que se participa, hace que TCP sea un **protocolo orientado a conexión**, esto es, que sea necesario realizar una negociación antes de intercambiar datos. A esa negociación inicial de TCP se le conoce como ***3-way handshake***, por los tres pasos que realiza (**Figura 2.13**).

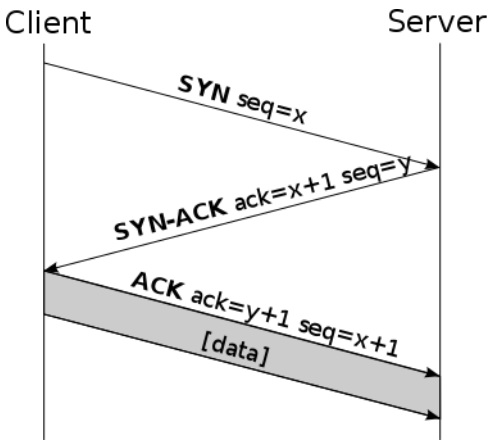



**Figura 2.13** – TCP *3-way handshake* (Fuente: Wikipedia)

En particular, el cliente es el que inicia la comunicación, enviado un segmento TCP sin datos, únicamente con el *flag* SYN activo y un número de secuencia aleatorio (x) al servidor. Si éste está dispuesto a recibir conexiones, enviará de vuelta otro segmento TCP sin datos, pero con los *flags* SYN y ACK, confirmando el número de secuencia del cliente (x+1) y enviando su propio número de secuencia aleatorio (y). Finalmente, el cliente debe enviar un segmento TCP con el *flag* ACK activo (ya sin el *flag* SYN), confirmando el número de secuencia del servidor (y+1). El último segmento del *3-way handshake* ya podría enviar datos, aunque normalmente estos se envían en un segmento posterior. Si el servidor no desea establecer una nueva conexión, responderá al primer segmento de SYN con otro segmento TCP sin datos y con los *flags* RST y ACK activos, confirmando el número de secuencia del cliente (x+1). Aunque en la mayoría de los protocolos es el cliente el que envía datos de nivel de aplicación primero, y luego el servidor responde, TCP permite enviar datos en ambos sentidos en cuanto se completa el *3-way handshake*.

En cuanto a la finalización de la conexión TCP, aunque es bastante habitual que los extremos cierren la conexión consecutivamente, realmente cada sentido de la conexión puede cerrarse de manera independiente. Para ello basta con enviar un segmento TCP con el *flag* FIN activo, que indica que el emisor no enviará más datos después de ese, y al que el otro extremo responderá con un *flag* ACK confirmando

su número de secuencia. Aunque éste podrá seguir enviando datos hasta que envíe su propio *flag* FIN para cerrar también su sentido de la conexión, y completar una desconexión ordenada. En caso de error (e.g. si la aplicación por encima de TCP se cierra inesperadamente), cualquiera de los extremos de la comunicación puede enviar un segmento con el *flag* de RST activo, que indica que la conexión debe cerrarse inmediatamente.

Por último, la cabecera TCP también permite incluir opciones, codificadas con una estructura TLV (Tipo-Longitud-Valor) y que permiten definir más parámetros de la conexión: como el tamaño máximo de los segmentos (*Maximum Segment Size*) que se pueden recibir, ampliar el tamaño máximo de la ventana de recepción (*Windows Scale*), incluir marcas de tiempo (*Timestamp*) en los segmentos para calcular el RTT (*Round Trip Time*) de la red, o implementar el mecanismo de confirmaciones selectivas (SACK). El tamaño de la cabecera TCP, incluidas sus opciones, se indica en el campo *Data Offset*, lo que también permite encontrar rápidamente dónde empiezan los datos de la aplicación.

## 2.8. Domain Name System (DNS)

Todos los protocolos de la pila TCP/IP utilizan direcciones IP binarias, pero los seres humanos no somos demasiado buenos recordando secuencias de números, así que enseguida surgió la necesidad de asociar nombres (mucho más memorizables y fáciles que reconocer que las direcciones IP) a las direcciones IP. Originalmente, cuando Internet consistía en una decena de nodos, simplemente se distribuía periódicamente un fichero HOSTS con los nombres asociados a todas las direcciones IP en uso. Ese mecanismo perdura hoy en día, y todos los sistemas operativos tienen un fichero HOSTS ('C:\Windows\System32\drivers\etc\hosts' en Windows y '/etc/hosts' en Linux/UNIX), que se consulta antes que el DNS, por lo que puede ser muy útil para asociar un nombre de dominio a una IP alternativa para simplificar la captura de tráfico.

Obviamente ese sistema dejó de escalar en cuanto Internet paso a tener miles de nodos, lo que motivó la creación del sistema de resolución de nombres DNS [RFC1035]. DNS es sistema de nombres distribuido y jerárquico, que permite a los responsables de un dominio publicar información relacionada con el mismo, como por ejemplo su dirección IP asociada. Un dominio no es nada más que una secuencia de etiquetas (que no distinguen entre mayúsculas y minúsculas), separadas mediante puntos (e.g. "www.example.com").

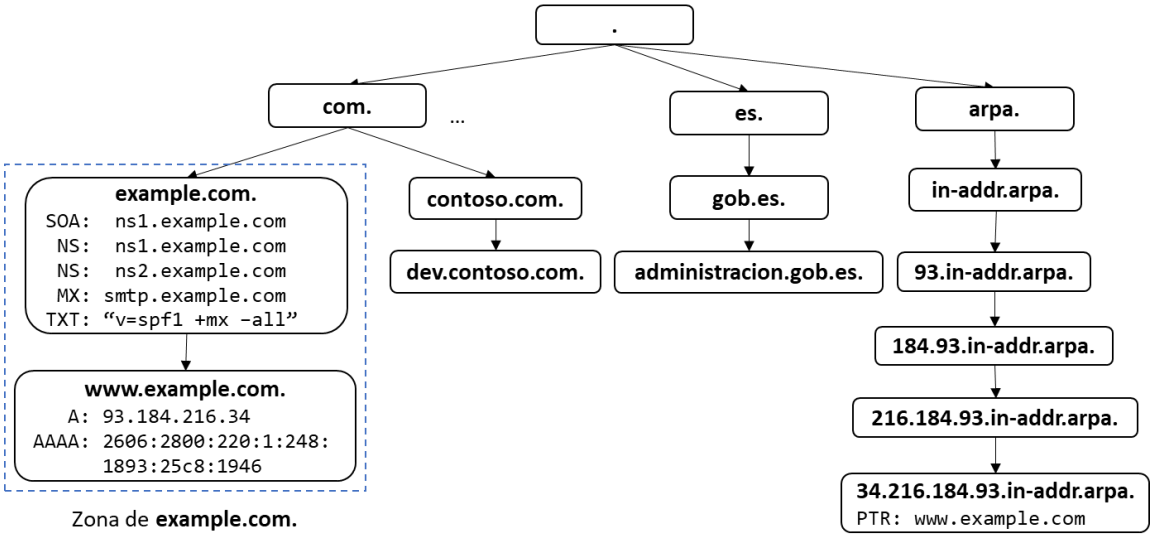



**Figura 2.14** – Jerarquía del Sistema de Nombres de Dominio (DNS)

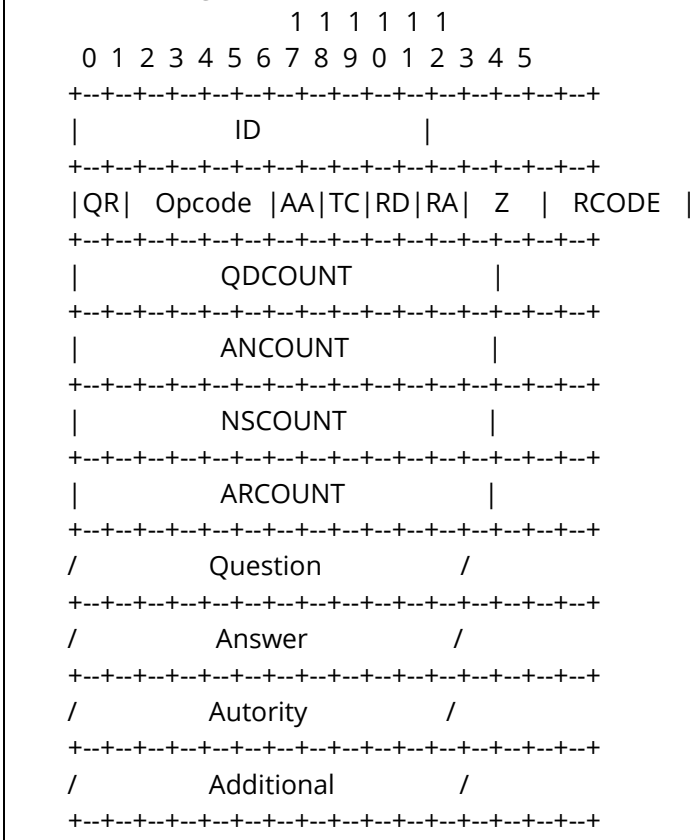
El sistema DNS tiene forma de árbol (**Figura 2.14**) y comienza en la raíz, que se representa como un punto '.'. Debajo de la raíz se encuentran los dominios TLD de primer nivel ("com", "org", "edu", "arpa", etc.) y los códigos de país ccTLD ("es", "uk", "us", etc.). Debajo se encuentran los dominios de segundo nivel (e.g. "example.com", "gob.es", "google.com", etc.), que suelen estar bajo el control de los diferentes tipos de organizaciones, y por debajo se pueden seguir creando subdominios, hasta llegar a un máximo de 253 caracteres. Cada nombre de dominio puede tener asociado uno o más **Registros de Recurso** (RR – *Resource Records*), que almacenan diferentes tipos de información. Los más importantes son:

- **A** (*Type*=1): Dirección IPv4 asociada a ese nombre de dominio.
- **AAAA** (*Type*=28): Dirección IPv6 asociada a ese nombre de dominio.
- **CNAME** (*Type*=5): Indica que este nombre de dominio es un alias de otro (e.g. www.example.com CNAME server1.example.com).
- **PTR** (*Type*=12): Apunta a otra parte del sistema DNS. Se utilizan para la resolución inversa de nombres (i.e. dirección IP → nombre de dominio asociado).
- **SOA** (*Type*=6): Marca el inicio de una autoridad de zona y define los parámetros comunes de la misma, como el servidor de nombres primario de la zona, una dirección de correo de contacto o cada cuanto tiempo deben replicar la información de zona los servidores de nombre secundarios.
- **NS** (*Type*=2): Servidor de nombres autoritativo de ese dominio. Todas las zonas deben tener al menos dos servidores de nombres: uno primario y el resto secundarios, que se sincronizan periódicamente con el primario.
- **MX** (*Type*=15): Servidor de correo entrante para ese dominio.

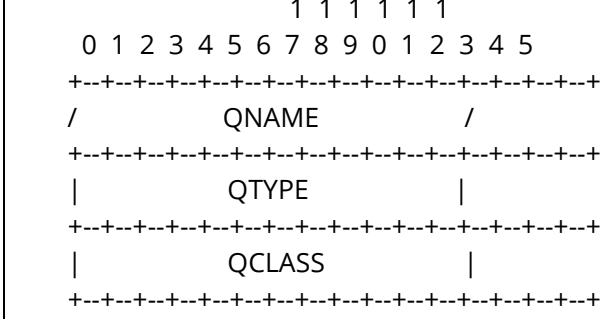
- **TXT** (*Type*=16): Permite publicar cadenas de texto. Algunas aplicaciones (e.g. SPF) los utilizan para publicar información en el DNS si necesidad de definir un nuevo tipo de registro.

Los dominios se agrupan en zonas, que están bajo el control una única entidad, aunque se puede delegar el control de cualquier subdominio a otra entidad (y por tanto se crea otra zona). Cada zona tiene uno o más servidores de nombres autoritativos, que son los responsables de mantener la información de la zona y responder a las consultas que les realizan otros clientes o servidores, mediante el protocolo DNS.

**A) DNS Message Format:**




**B) Question Section Format:**




**C) Resource Record Format:**

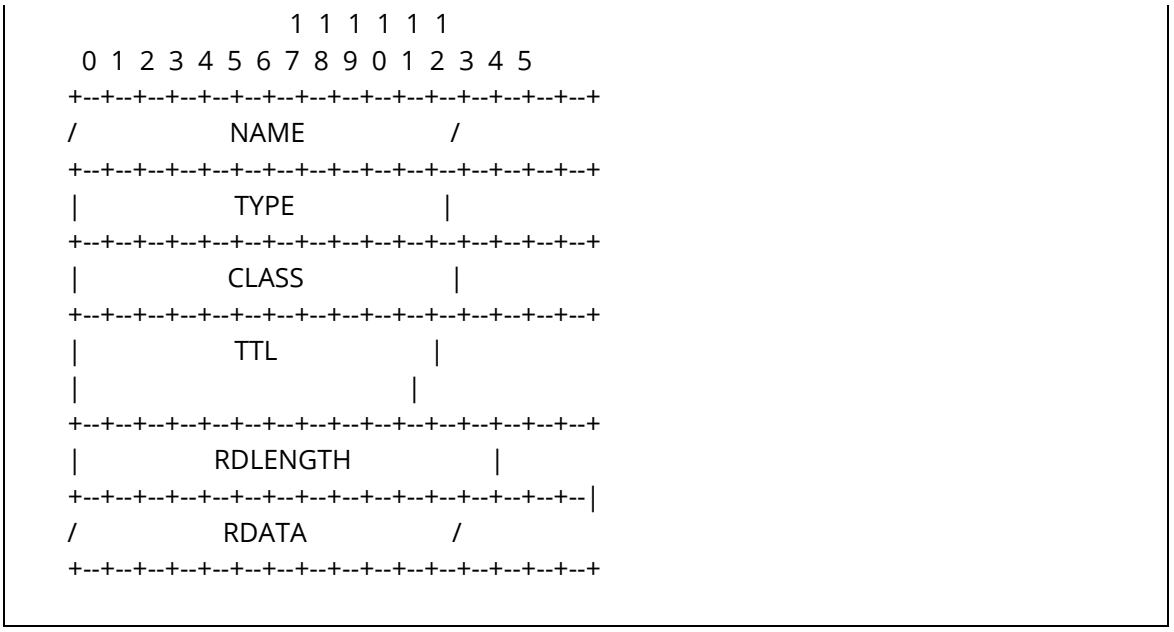



**Figura 2.15** – Formato de los mensajes y secciones del protocolo DNS [RFC1035]

DNS es un protocolo cliente-servidor basado en UDP, y que por defecto está limitado a mensajes de 512 octetos, aunque la extensión EDNS [RFC6891] permite ampliar este tamaño. Los mensajes DNS (**Figura 2.15**) están formados por una cabecera fija, y varias secciones de tamaño variable, que pueden incluir múltiples registros de recurso (RRs). En DNS solo hay dos tipos de mensajes: peticiones (QR=0; *Opcode*=0) y respuestas (QR=1, *Opcode*=0).

Las peticiones recursivas (RD=1) tienen un identificador de 16 bits, que el servidor debe incluir en su respuesta, y suelen incluir una única consulta (QDCOUNT=1), que incluye el nombre de dominio que se quiere consultar (QNAME) y el tipo de registro que se solicita (QTYPE), porque en Internet la clase siempre es IN (QCLASS=1).

Las respuestas usan el campo RCODE para indicar si se ha producido algún error, como por ejemplo si el dominio no existe (*NXDomain*). Además, los *flags* permiten especificar si la repuesta viene del servidor autoritativo de la zona (AA=1), si el servidor soporta consultas recursivas (RA=1), o si la respuesta está truncada (TC=1), en cuyo caso el cliente debería reintentar la consulta con TCP, que no tiene límite de tamaño. Las respuestas pueden copiar la sección de consulta, y utilizan la sección *Answer* para incluir los RRs solicitados si los conocen o, en caso contrario, envían información sobre los servidores de nombres autoritativos del dominio (*Authority*). Las respuestas DNS pueden incluir además información adicional (*Additional*), como las direcciones IP de los servidores de nombres autoritativos. Para mejorar el rendimiento de DNS, los RRs obtenidos pueden cachearse durante TTL segundos.

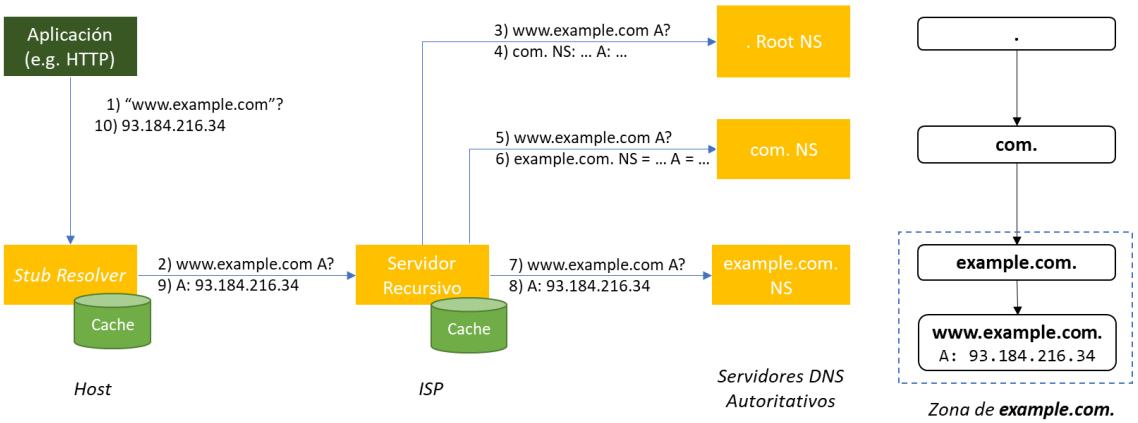



**Figura 2.16** – Ejemplo de proceso de resolución de nombres DNS

Cuando una aplicación desea resolver un nombre DNS (**Figura 2.16**), normalmente no implementa directamente el protocolo DNS, sino que consulta a un servicio local denominado ***Stub Resolver***, que es el que realmente implementa la parte cliente del protocolo DNS, y envía la consulta DNS al **Servidor de Nombres Recursivo** que tiene configurado, y que normalmente es un servicio que proporciona la organización o el proveedor de acceso a Internet de los usuarios. Se denomina recursivo porque es capaz de responder a una consulta DNS, incluso si no la sabe y necesita contactar con más servidores DNS hasta encontrar la respuesta. Y es que, como el sistema DNS es un sistema distribuido, para poder resolver un nombre DNS, primero hay que localizar al **Servidor de Nombres Autoritativo** que se encarga de esa zona. Para ello, como el sistema DNS es un sistema jerárquico, los servidores DNS recursivos pueden empezar preguntando a uno de los 13 **Servidores de Nombres Raíz** que hay en Internet (aunque realmente está replicados en cientos de localizaciones por todo el mundo). Éste le indicará al servidor recursivo que no conoce la respuesta, pero le indicará el servidor de nombres (NS) asociado al dominio de primer nivel (e.g. "com."), así como su dirección IP (A). Así que el servidor recursivo volverá a preguntar a ese servidor DNS, que típicamente tampoco conoce la respuesta, por lo que devuelve al servidor recursivo el servidor de nombres (NS) del subdominio consultado (e.g. "example.com"). El servidor recursivo seguirá preguntando a servidores de nombres autoritativos, hasta que uno de ellos sea el responsable de la zona que contiene el nombre consultado, y éste por fin le devolverá la respuesta deseada (e.g. la dirección IPv4 93.184.216.32). El servidor de nombres recursivo almacenará toda la información recopilada, incluyendo la respuesta y la información sobre todos los servidores autoritativos en su cache, de forma que, cuando un cliente vuelva a preguntar por alguno de esos dominios, pueda consultar directamente al servidor de nombres autoritativo más apropiado (en lugar de repetir todo el camino desde la raíz), y enviará la respuesta al cliente DNS (que también puede tener una caché local), y éste le enviará finalmente la información a la aplicación.

El *malware* utiliza a veces el tráfico DNS para exfiltrar información (e.g. a través de recursos TXT o codificando la información en las consultas que realiza) y sobre todo para localizar sus servidores de *Command and Control* (C&C/C2), de forma que puedan cambiarlos en caso de que su dirección IP sea descubierta. Para evitar que las organizaciones también filtren el dominio DNS del C2, algunas familias de *malware* incluyen un *Domain Generated Algorithm* (DGA) que, en función del tiempo, es capaz de generar dominios DNS pseudo-aleatorios, de forma que cada cierto tiempo el servidor de C2 se mueve a un dominio DNS diferente, por lo que es mucho más difícil filtrarlo.

## 2.9. Transport Layer Security (TLS)

*Transport Layer Security* (TLS), o *Secure Sockets Layer* (SSL) como se llamaban las primeras versiones del protocolo, es el protocolo seguro más importante de Internet. Es un protocolo cliente-servidor que funciona encima de TCP (mientras que DTLS [RFC6347] funciona sobre UDP), garantiza la confidencialidad e integridad de las comunicaciones que la usan, y permite autenticar mediante certificados digitales al servidor (y opcionalmente al cliente), por lo que también evita ataques de *Adversary-in-the-Middle* (AitM) si se utiliza una lista de autoridades de certificación confiable.

TLS ha tenido múltiples versiones (SSL 2.0, SSL 3.0, TLS 1.0, TLS 1.1, TLS 1.2 y TLS 1.3), que han ido resolviendo los problemas de seguridad de las versiones anteriores, y añadiendo algoritmos criptográficos cada vez más seguros. Actualmente las últimas dos versiones de TLS (v1.2 [RFC5246] y v1.3 [RFC8446]) se consideran seguras, y la única forma de descifrar el tráfico intercambiado es mediante la colaboración de alguno de los extremos de la comunicación.

TLS es un protocolo bastante complejo, que de hecho tiene varios protocolos o capas internas. La *Record Layer* es la capa inferior de TLS y se encarga de delimitar los mensajes que envían los diferentes protocolos de TLS en estructuras de tipo TLV (Tipo-Longitud-Valor), y que también incluyen la versión de TLS en uso. Esta capa es la que se encarga de cifrar y garantizar la integridad de los datos mediante códigos de autenticación (HMAC). Para ello utiliza un conjunto de claves de cifrado y de firmado, que son diferentes en cada sentido y se negocian de nuevo en cada sesión TLS.

Por encima de la *Record Layer* se encuentran los cuatro protocolos de TLS: el *Change Cipher Spec Protocol*, el *Alert Protocol*, el *Handshake Protocol*, y el *Application Data Protocol*. Los dos primeros son muy sencillos, puesto que se limitan a un único mensaje: que sirve para indicar que a partir de ese momento se van a cifrar todos los datos que envíe la *Record Layer* (*Change Cipher Spec*), y para notificar problemas

en la sesión TLS (*Alert*), lo que suele implicar la finalización de la misma. El *Application Data Protocol* no es más que el protocolo de nivel de aplicación que envía datos mediante TLS. Así que el protocolo más importante de TLS es el *Handshake Protocol*, que es el que se emplea para negociar todas las claves que usa la *Record Layer* y que se ejecuta tan pronto como se completa el *3-way handshake* de la conexión TCP subyacente.

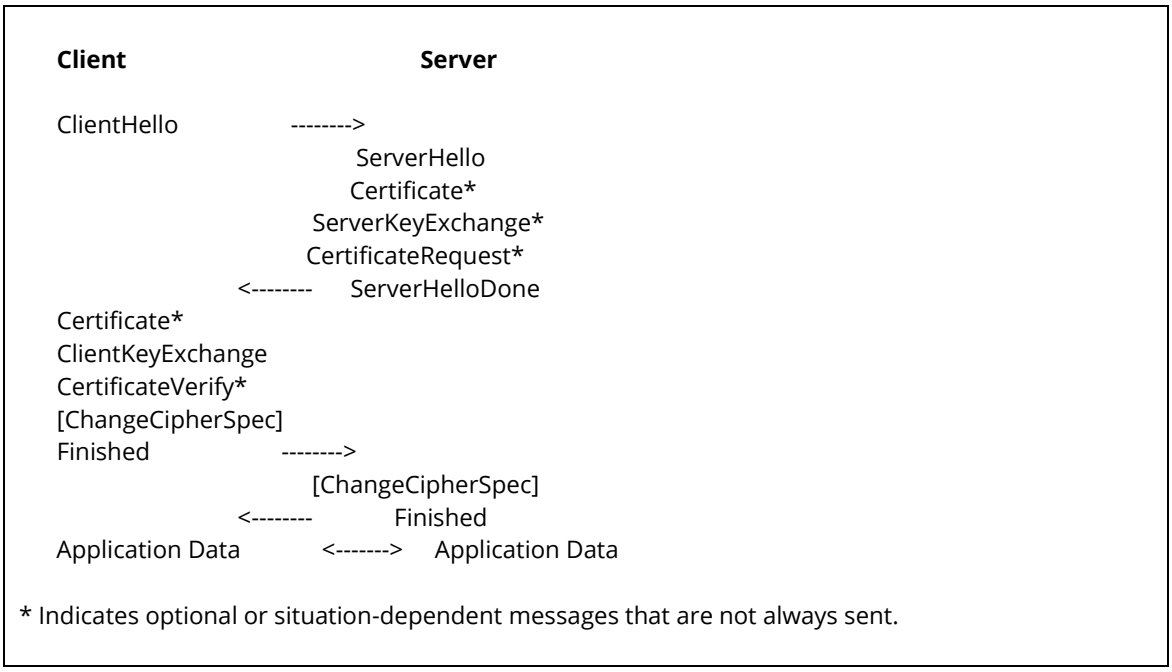



**Figura 2.17** - Establecimiento (*handshake*) de una sesión TLSv1.2 [RFC5246]

La **Figura 2.17** muestra todos los mensajes que se pueden llegar a enviar en un *handshake* completo de una sesión TLS v1.2:

- ***ClientHello***: Este mensaje contiene la versión de TLS más alta que implementa el cliente (e.g. v1.2), así como la lista de todos los *ciphersuites* y extensiones que soporta. Un ***ciphersuite*** TLS (e.g. TLS_DHE_RSA_WITH_AES_128_GCM_SHA256) es un conjunto de algoritmos criptográficos que se emplean para negociar las claves (e.g. DHE), firmar digitalmente (e.g. RSA), cifrar (e.g. AES_128_GCM) y como función hash (e.g. SHA256) de las firmas digitales y HMAC. Las extensiones TLS permiten añadir funcionalidades opcionales al protocolo, tales como el uso de algoritmos de curva elíptica para el intercambio de claves o de firma digital. En particular, hay dos extensiones muy importantes para la captura y análisis de sesiones TLS: la extensión ***Server Name Indication*** **(SNI)** y la extensión ***Aplication Layer Protocol Negotation*** **(ALPN)**, que indican, respectivamente, el nombre DNS del servidor TLS con el que se quiere conectar el cliente (por si hay varios servidores virtuales compartiendo la misma dirección IP), y los protocolos de

nivel de aplicación soportados por el cliente. La importancia de estas extensiones radica en que el mensaje *ClientHello* no está cifrado, por lo que todos sus campos se envían en texto claro, y es posible conocer el servidor TLS con el que se quiere comunicar un cliente TLS y el protocolo de aplicación superior, incluso cuando todos los datos intercambiados por el nivel de aplicación estén cifrados. El conjunto de *ciphersuites* y extensiones soportadas también permite intentar identificar la librería criptográfica que emplea el cliente (e.g. OpenSSL 1.1.1).

- ***ServerHello***: Cuando el servidor TLS recibe el mensaje *ClientHello* busca los *ciphersuites* y extensiones del cliente que también soporte el servidor, elige las más seguras, y envía este mensaje para indicar al cliente qué versión de TLS, *ciphersuite* y extensiones se van a emplear en esa sesión TLS.
- ***Certificate***: Este mensaje contiene el certificado X.509 del servidor TLS, incluyendo los certificados de todas las autoridades de certificación (CAs) intermedias hasta llegar (sin incluir) a una de las autoridades de certificación raíz de las que se fía el cliente. De forma que éste pueda comprobar la validez de toda la cadena de certificados, y por tanto fiarse de la identidad del servidor (porque está validada por una CA confiable).
- ***CertificateRequest***: Además de autenticar al servidor, TLS permite, de manera opcional, autenticar al cliente mediante otro Certificado. Este mensaje sirve para solicitar al cliente que envíe al servidor su certificado en un mensaje *Certificate*, indicando las autoridades de certificación (CAs) de las que se fía el servidor.
- ***ServerKeyExchange***: Las primeras versiones de SSL y TLS utilizaban la clave RSA pública del certificado del servidor para implementar el intercambio de claves. El cliente simplemente generaba de manera aleatoria un secreto maestro (denominado ***Pre-Master Secret Key***), del que se derivan el resto de claves de la sesión SSL/TLS, y lo cifraba con la clave pública RSA del servidor. De esta forma además se comprobaba que el servidor tuviese la clave privada asociada, porque esa era la única forma de obtener el secreto maestro y por tanto las claves de sesión necesarias para descifrar el tráfico. El problema de este mecanismo de intercambio de claves es que, si en el futuro alguien consigue comprometer la clave privada del servidor, sería capaz de descifrar todo el tráfico que cualquier cliente se ha intercambiado con el servidor utilizando ese certificado. Así que en las últimas versiones de TLS (y obligatoriamente en TLSv1.3) la generación del *Pre-Master Secret Key* se puede realizar mediante un protocolo de intercambio de claves Diffie-Hellman Efímero (DHE) o basado en curva elíptica (ECDH). Este mensaje envía los parámetros de DHE o ECDH del servidor al cliente, que deben estar firmados con la clave privada del servidor para demostrar la posesión del certificado (el cliente comprueba la firma con la clave pública del certificado).

- ***ServerHelloDone***: Una vez que el servidor termina de enviar sus mensajes (*ServerHello*, *Certificate* y opcionalmente *CertificateRequest* y/o *ServerKeyExchange*), utiliza este mensaje para indicar al cliente que es su turno.
- ***ClientKeyExchange***: Este mensaje completa el intercambio de claves con el servidor, bien enviando el *Pre-Master Secret Key* cifrado con la clave pública que aparece en el certificado del Servidor, o bien enviando los parámetros DHE o ECDHE del cliente.
- ***Certificate*** y ***CertificateVerify***: Cuando el servidor solicita al cliente un certificado con el mensaje *CertificateRequest*, el cliente debe enviar su certificado en un mensaje *Certificate*, que debe estar firmado por alguna de las CAs indicadas por el servidor o incluir una cadena de certificados hasta alguna de ellas. El mensaje *CertificateVerify* está firmado digitalmente con la clave privada del cliente, para demostrar así su posesión.
- ***ChangeCipherSpec***: En cuanto el cliente genera el *Pre-Master Secret Key* (*ClientKeyExchange*) o lo negocia mediante DHE o ECDHE (*ServerKeyExchange*), puede generar todas las claves que necesita la *Record Layer* para cifrar y autenticar sus mensajes. Así que el cliente utiliza el mensaje *ChangeCipherSpec* (aunque formalmente no forma parte del *Handshake Protocol*) para indicar que, a partir de ese momento, todos los mensajes que enviará el cliente están cifrados con las claves negociadas. Del mismo modo, en cuanto el Servidor recibe el *ClientKeyExchange* del cliente, es capaz de obtener el *Pre-Master Secret* y de ahí todas las claves de cifrado y autenticación de la sesión, de modo que también utiliza el mensaje *ChangeCipherSpec* para indicar que a partir de ese momento todos sus mensajes también van a estar cifrados con esas claves.
- ***Finished***: Estos son los primeros mensajes de la sesión TLS que están cifrados y autenticados. Eso quiere decir que todos los mensajes anteriores se transmiten en claro y podrían haber sido modificados por un atacante (e.g. para negociar una versión de SSL vulnerable). El mensaje *Finished* se utiliza para enviar una suma de comprobación de todos los mensajes del protocolo *Handshake* que se han recibido (que no puede ser modificada ya que el mensaje *Finished* está autenticado) de forma que el otro extremo compruebe que coincide exactamente con los mensajes que ha enviado él y por tanto que nadie los ha modificado.

Una vez que ambos extremos han comprobado sus mensajes *Finished*, se establece la sesión TLS y puede empezar el intercambio seguro de datos de nivel de aplicación (e.g. HTTP).

El problema de la negociación TLS es que es bastante largo (2x RTT), requiere transmitir uno o más certificados (que pueden ocupar varios Kilobytes) y el uso de

criptografía asimétrica para cifrar, firmar y/o negociar claves. Así que TLS también define un mecanismo de negociación de sesión TLS rápido, denominado *Session Resumption*.

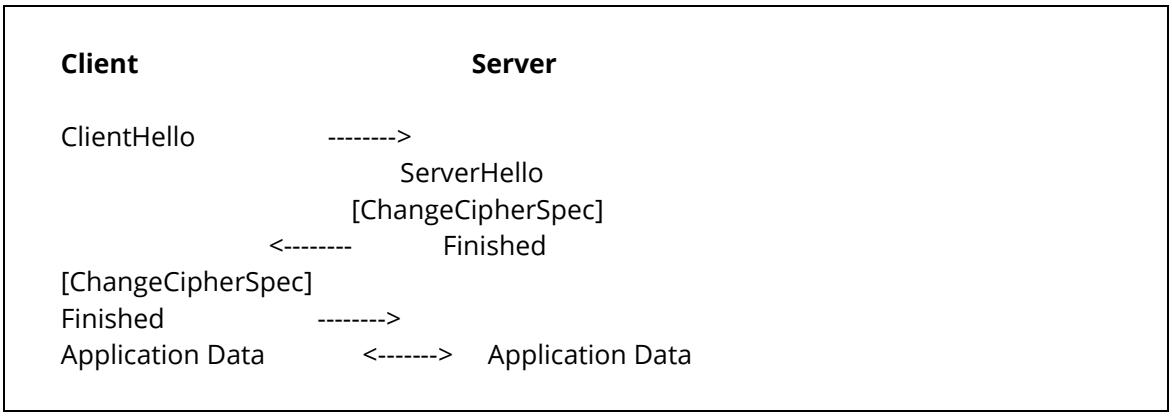



**Figura 2.18** – Resumen de una sesión TLSv1.2 [RFC5246]

La idea del TLS *Session Resumption* es reutilizar las claves simétricas ya negociadas en una sesión TLS anterior, de forma que se pueda reducir mucho el tiempo y la computación necesaria para el completar el protocolo de *Handshake*. Existen un par de formas para implementar el *Session Resumption*. En la primera, tanto el cliente como el servidor almacenan las claves negociadas anteriormente y las asocian con un *SessionID* que establece el servidor en su mensaje *ServerHello*. De forma que cuando el cliente vuelva a conectarse con el servidor, envía ese *SessionID* en su mensaje *ClientHello* para indicar que desea resumir esa sesión TLS. Si el servidor todavía se acuerda de las claves de esa sesión, responde con un mensaje *ServerHello* con el mismo valor de *SessionID* y, como no es necesario autenticarse frente al cliente (porque ya lo hizo en la sesión anterior) ni negociar claves de sesión, envía inmediatamente el mensaje *ChangeCipherSpec,* y empieza a utilizar las claves anteriores para cifrar y firmar el mensaje *Finished* que sirve para comprobar que los mensajes *ClientHello* y *ServerHello* no han sido alterados. El cliente hace exactamente lo mismo (incluso si el servidor requiere la autenticación del cliente mediante certificado), por lo que el *Handshake* de la sesión resumida se limita a 1.5 RTTs y no hay que enviar ningún certificado ni utilizar ninguna operación de criptografía asimétrica. Si el servidor no se acuerda de las claves del *SessionID* solicitado por el cliente, simplemente genera un nuevo *SessionID* y se ejecuta un *Handshake* TLS completo.

La única pega de este mecanismo de resumen mediante *SessionID* es que requiere que los servidores almacenen las claves de sesión de todos sus clientes, y sobre todo que sean capaces de recuperarlas cuando el cliente lo solicite. Lo cual es un problema en granjas de servidores, donde el cliente puede contactar con un servidor diferente. Por esa razón se creó un nuevo mensaje de servidor, denominado *NewSessionTicket* [RFC5077], que actúa como una especie de *Cookie*

HTTP para almacenar información cifrada en el cliente, y una extensión para los mensajes *ClientHello* y *ServerHello* para indicar que ambos extremos soportan este mecanismo. El Mensaje *NewSessionTicket* lo envía el servidor al cliente justo antes de su mensaje *ChangeCipherSpec* (**Figura 2.19.A**) y contiene las claves de sesión que acaba de negociar con el cliente (i.e. *ClientKey Exchange*), cifradas con una clave simétrica que solo conocen él y resto de servidores de la granja.

De esta forma no hace falta que el servidor se acuerde de las claves de sesión de ese cliente, porque la próxima vez que el cliente se conecte al servidor y quiera resumir la sesión (**Figura 2.19.B**), solo tiene que incluir el valor del *NewSessionTicket* que recibió del servidor en la extensión *SessionTicket* de su *ClientHello*. Así que el Servidor puede descifrar las claves de sesión con su clave secreta y resumir la sesión como si se utilizase el mecanismo de *SessionID* básico, aunque puede volver a enviar un mensaje *NewSessionTicket* para renovar el estado del cliente.

**A)** **Client** **Server**

ClientHello
(SessionTicket extension) -------->
ServerHello
(empty SessionTicket extension)
Certificate*
ServerKeyExchange*
CertificateRequest*
<-------- ServerHelloDone
Certificate*
ClientKeyExchange
CertificateVerify*
[ChangeCipherSpec]
Finished -------->
NewSessionTicket
[ChangeCipherSpec]
<-------- Finished
Application Data <-------> Application Data

**B)** **Client** **Server**

ClientHello
(SessionTicket extension) -------->
ServerHello
(empty SessionTicket extension)
NewSessionTicket
[ChangeCipherSpec]
<-------- Finished

```
[ChangeCipherSpec]
Finished                -------->
Application Data        <-------->  Application Data
```

**Figura 2.19** – A) Negociación y B) Resumen de una sesión TLSv1.2 con *SessionTicket* [RFC5077]

Como TLS trabaja por encima de TCP, es capaz de proteger los datos de aplicación, pero no la conexión TCP subyacente. Por lo tanto, un atacante que sea capaz de observar los números de secuencia de la conexión TCP, puede cerrarla prematuramente falsificando un TCP FIN o un RST, lo que truncaría la sesión TLS. Para evitar este ataque, y señalizar que la sesión TLS se ha completado correctamente antes de cerrar la sesión TLS, las implementaciones de TLS generan un mensaje de *Alert* cifrado con el código de error *Close Notify*, justo antes de cerrar la conexión subyacente. Así que, aunque sea contraintuitivo, las sesiones TLS que terminan correctamente deben incluir un mensaje *Alert* justo antes de cerrar la conexión TCP subyacente (que por otro lado es lo que ocurriría si se detecta un error grave en la sesión TLS).

## 2.10. HyperText Transfer Protocol (HTTP)

HTTP es el protocolo empleado para navegar por la *World Wide Web* (WWW), puesto que es el protocolo que permite a los navegadores web descargar el código de las páginas web (codificadas en lenguaje HTML), de sus recursos asociados (imágenes, código *javascript*, hojas de estilo CSS, ficheros, etc.), así como interactuar con ellas (e.g. enviar los datos de un formulario al servidor web). Los recursos web se identifican mediante URLs (*Uniform Resource Locator*) [RFC1738] (**Figura 2.20**), lo que permite identificarlos y acceder a ellos, incluso si están en otro servidor.

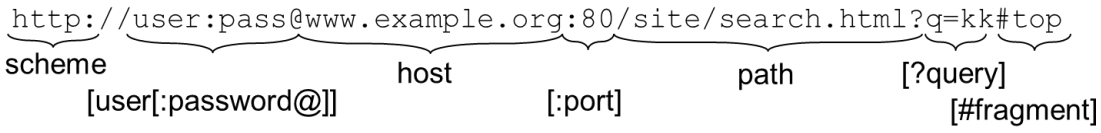



**Figura 2.20** – Formato de una URL

HTTP es un protocolo-cliente servidor que funciona sobre TCP (puerto 80). El protocolo HTTPS no es más que el protocolo HTTP sobre TLS/TCP (puerto 443). El protocolo HTTP es un protocolo orientado a texto en el que el cliente envía peticiones, típicamente solicitando recursos, y el servidor responde con el recurso, más ciertas cabeceras de metainformación.

Las peticiones HTTP (**Figura 2.21**) están formadas por una línea de petición, seguida por varias cabeceras. Todas ellas son líneas de texto que terminan con los símbolos

ASCII de salto de línea y retorno de carro (\r\n). Algunos tipos de peticiones HTTP (POST, PUT) pueden incluir también un cuerpo de mensaje con datos, que se separa de las cabeceras mediante una línea en blanco.

```
GET / HTTP/1.1\r\n
Host: www.example.com\r\n
User-Agent: Wget/1.20.3 (linux-gnu)\r\n
Accept: */*\r\n
Accept-Encoding: identity\r\n
Connection: Keep-Alive\r\n
\r\n
```

**Figura 2.21** – Ejemplo de petición HTTP

La línea de petición (en negrita) define el método HTTP utilizado, el *path, query y fragment* del recurso solicitado (o la URL completa si se utiliza un *proxy* web), y la versión del protocolo más alta que soporta el cliente (separados con espacios). La primera versión de HTTP/1.0 solo permitía descargar un único recurso por conexión TCP (puesto que utilizaba el fin de la conexión para señalizar el final de la descarga del recurso), por lo que fue rápidamente reemplazado por la versión HTTP/1.1 [RFC2616], que permite utilizar la misma conexión TCP para enviar múltiples peticiones y respuestas, una tras otra. HTTP/2.0 [RFC7540], aunque es compatible con los mensajes y cabeceras de sus antecesores, es un protocolo binario bastante más complejo, que define múltiples canales para poder enviar y recibir múltiples peticiones y respuestas simultáneamente sobre la misma conexión TCP.

Los principales métodos de HTTP son:

- ***GET***: Solicita la descarga de un recurso.
- ***POST***: Solicita añadir información (incluida en el cuerpo de la petición) a un recurso ya existente. Se utiliza por ejemplo para enviar los datos de un formulario al servidor.
- ***PUT***: Solicita crear un recurso (incluido en el cuerpo de la petición).
- ***DELETE***: Solicita borrar el recurso indicado.
- ***HEAD***: Solicita descargar únicamente la metainformación de un recurso (i.e. sus cabeceras HTTP), no el recurso en sí.
- ***OPTIONS***: Solicita la lista de métodos que se pueden ejecutar sobre el recurso.
- ***CONNECT***: Este método se emplea para comunicarse con un servidor HTTPS a través de un *proxy* HTTP. En lugar de un recurso, se indica al *proxy* el nombre DNS o la dirección IP y el puerto del servidor al que debe conectarse.

Aunque para la navegación web normalmente solo se utilizan los métodos *GET*, *HEAD* y *POST*, las APIs REST [REST] están basadas en el protocolo HTTP y utilizan todos los métodos para realizar operaciones sobre los recursos (que también se representan con URLs).

Además de la línea de petición, las cabeceras más habituales que pueden aparecer en una petición HTTP son:

- ***Accept*:** Lista de tipos de contenidos soportados por el cliente (e.g. "text/html, text/plain").
- ***Accept-Encoding*:** Lista de formatos de compresión soportados por el cliente (e.g. "identity" o "gzip").
- ***Accept-Language*:** Lista de lenguajes del usuario (e.g. "es-ES, en;q=0.8").
- ***Authorization / Proxy-Authorization*:** Tipo de autenticación (Basic | Digest | Bearer) y autenticador para el servidor/*proxy* en Base64 (e.g. "Basic dXNlcjpwYXNzd29yZA==").
- ***Cache-control*:** Directivas para la cache superiores (e.g. "no-cache").
- ***Connection*:** Permite controlar la conexión TCP subyacente (e.g. "keep-alive").
- ***Cookie*:** Devuelve una *cookie* al servidor (e.g. "SID=31d4d96e407aad42; lang=es-es")
- ***Content-Length***: Longitud del cuerpo de la petición en bytes.
- ***Content-Type*:** Tipo MIME del cuerpo de la petición (e.g. "application/x-www-form-urlencoded")
- ***Host*:** Nombre DNS del servidor (virtual).
- ***If-None-Match*:** Petición condicional basada en la versión del contenido (i.e. cabecera de respuesta ETag).
- ***If-Modified-Since***: Petición condicional basada en la fecha de modificación del recurso (i.e. cabecera de respuesta Last-Modified).
- ***Max-Forwards*:** Número máximo de saltos a través de *proxies* intermedios.
- ***Range*:** Permite solicitar la descarga parcial del recurso (e.g. "bytes=1024-2048")
- ***Referer*:** URL de la página web anterior.
- ***User-Agent*:** Identificador del navegador web (e.g. "Mozilla/5.0 (Windows NT 10.0; WOW64; rv:42.0) Gecko/20100101 Firefox/42.0")
- ***Via*:** Lista de *proxies* (<*versión HTTP*> <*proxy*> <*agente*>) por la que ha pasado la petición (e.g. "1.1 proxy.example.org:8080 (squid)").

Las respuestas HTTP (**Figura 2.22**) también están formadas por una línea de respuesta, seguida de múltiples cabeceras, y separadas del cuerpo de la respuesta (que puede ser un contenido en formato texto o binario) mediante una línea en blanco (\r\n).

```
HTTP/1.1 200 OK\r\n
Server: ECS (mic/9AF8)\r\n
Date: Wed, 06 Jan 2021 16:22:43 GMT\r\n
Etag: "3147526947+ident"\r\n
Last-Modified: Thu, 17 Oct 2019 07:18:26 GMT\r\n
Age: 495344\r\n
Cache-Control: max-age=604800\r\n
Expires: Wed, 13 Jan 2021 16:22:43 GMT\r\n
X-Cache: HIT\r\n
Content-Type: text/html; charset=UTF-8\r\n
Content-Length: 1256\r\n
\r\n
<!doctype html>
<html>
<head>
   <title>Example Domain</title>

   <meta charset="utf-8" />
   <meta http-equiv="Content-type" content="text/html; charset=utf-8" />
   <meta name="viewport" content="width=device-width, initial-scale=1" />
   ...
</head>
<body>
...
</body>
</html>
```

**Figura 2.22** – Ejemplo de respuesta HTTP

La línea de respuesta (en negrita) está formada por la versión HTTP soportada por el servidor, un código numérico y un mensaje de error (todo ello separado mediante espacios).

Los códigos y mensajes de error HTTP más comunes son:

- **200 *OK***: La petición ha tenido éxito, el recurso solicitado está incluido en el cuerpo del mensaje.
- **302 *Moved Permanently***: El recurso solicitado ya no se encuentra en esta localización. La cabecera Location: indica la nueva URL donde se encuentra el recurso.
- **304 *Not Modified***: El recurso de la petición condicional no ha cambiado respecto al versión (If-None-Match) o fecha (If-Modified-Since) indicadas.
- **400 *Bad Request***: La petición no se puede procesar porque tiene un error de sintaxis.
- **401 *Unauthorized***: El recurso solicitado requiere autenticación. El mecanismo de autenticación requerido por el servidor se indica en la cabecera WWW-Autheticate.
- **404 *Not Found***: El recurso solicitado no existe.

- **500 *Internal Server Error***: La petición no se ha podido procesar por un error interno del servidor.

Las cabeceras más habituales que pueden aparecer en una respuesta HTTP son:

- ***Age*:** Número de segundos que lleva el objeto en la cache.
- ***Allow*:** Métodos soportados por el recurso, solicitados con el método OPTIONS (e.g. "GET, HEAD").
- ***Cache-control*:** Directivas para el control de las caches inferiores (e.g. "private, max-age=3600").
- ***Connection*:** Permite controlar la conexión TCP subyacente (e.g. "close")
- ***Content-Encoding*:** Mecanismo de compresión del cuerpo del mensaje (e.g. "gzip")
- ***Content-Language*:** Lenguaje empleado en el recurso (e.g. "es-es").
- ***Content-Length*:** Longitud en bytes del cuerpo del mensaje.
- ***Content-Range*:** Rango de bytes descargado del recurso y tamaño total del mismo (e.g. "bytes 1024-2048/4096").
- ***Content-Type*:** Tipo MIME y codificación del cuerpo del mensaje (e.g. "text/html; charset=UTF-8").
- ***Date*:** Fecha cuando se envió la respuesta (e.g. "Tue, 24 Nov 2015 13:07:31 GMT").
- ***ETag*:** Identifica la versión del recurso, por ejemplo, realizando un hash de su contenido (e.g. "737060cd8c284d8af7ad3082f209582d").
- ***Expires*:** Fecha cuando el recurso dejará de ser válido (e.g. "Tue, 24 Nov 2015 14:00:00 GMT").
- ***Last-Modified*:** Fecha de la última modificación realizada (e.g. "Tue, 24 Nov 2015 13:00:00 GMT")
- ***Location*:** URL con la nueva localización del objecto (para respuestas con código 302 *Moved Permanently*).
- ***Refresh*:** Redirige el cliente a una nueva URL tras un cierto número de segundos (e.g. "5; url=https://www.example.com")
- ***Server / X-Powered-By*:** Identifican el servidor HTTP y las librerías que usa (e.g. "Server: Apache/2.4.1 (Unix)", "X-Powered-By: PHP/5.4.0").
- ***Set-Cookie*:** Establece una cookie en el cliente, que éste debería enviar de vuelta cada vez que se conecte al servidor (e.g. "SID=31d4d96e407aad42; max-age=86400; domain=.example.com; path=/; secure; httpOnly")
- ***WWW-Authenticate*:** Mecanismo de autenticación requerido para acceder al recurso (e.g. "Basic").

Las cabeceras que comienzan por **X-** (e.g. X-Powered-By) no están estandarizadas, lo que permite a las aplicaciones definir cabeceras nuevas.

HTTP es un protocolo sin estado. Eso quiere decir que el servidor puede procesar las peticiones de los clientes de manera independiente, sin que exista alguna noción

a nivel HTTP de que un conjunto de peticiones pertenezca a una misma "sesión de navegación". De hecho, cuando los recursos de un servidor requieren autenticación HTTP, todas las peticiones del cliente tienen que incluir la cabecera Authorization con sus credenciales. Sin embargo, muchas aplicaciones web por encima de HTTP sí necesitan un concepto de sesión, por ejemplo, para comprobar si un cliente ya se ha autenticado, o para recordar los productos que ha introducido en el carro de la compra. Para ello, HTTP proporciona un mecanismo de *Cookies* a la aplicación, que permite al Servidor enviar al Cliente (en la cabecera *Set-Cookie*) una o más Cookies con información opaca, y que el cliente debe devolver la próxima vez que envíe una petición al mismo servidor (en la cabecera Cookie). De este modo, a un cliente autenticado se le puede enviar una *Cookie* de sesión (con un valor aleatorio y/o firmado por el servidor) que sirve de índice en la base de datos donde se almacena la información de ese cliente, o simplemente incluir toda esa información en la propia *Cookie* (que puede ocupar hasta 4 KiB), idealmente de una manera cifrada y firmada.

### 2.10.1. Proxies HTTP

Un aspecto muy interesante de HTTP es que, a pesar de ser un protocolo cliente-servidor, permite que las comunicaciones atraviesen cero o más *proxies*. Hace unos años, los *proxies* web se desplegaban típicamente en el lado de los clientes, para que estos almacenaran en una cache los contenidos más populares, y ahorrar así capacidad en el acceso a Internet de las organizaciones. Sin embargo, la gran cantidad contenido dinámico y personalizado de la web 2.0 hace que hoy en día la mayoría de los ***proxies* del lado cliente** se desplieguen por motivos de seguridad: para evitar que los equipos de los empleados salgan directamente a Internet, y para monitorizar los contenidos que intercambian (e.g. con un antivirus o para limitar el acceso a sitios web maliciosos o con contenidos cuestionables).

Normalmente los usuarios son conscientes de la existencia de los *proxies* del lado cliente, y configuran sus navegadores para que los utilicen para salir a Internet. En ese caso, el navegador web establece una conexión TCP con el *proxy* configurado (por defecto usando el puerto TCP/8080), y le envía todas las peticiones HTTP que genera, tal cual haría con el servidor web, con la salvedad que en lugar de enviar URLs relativas (e.g. "/index.html") en la línea de petición, se envían URLs absolutas (e.g. http://www.example.com:80/index.html) para indicarle al *proxy* el servidor web al que debe reenviar la petición. El *proxy* establece entonces una conexión TCP con el servidor web y le reenvía la petición del cliente (añadiendo una cabecera Via para indicar que ha pasado por él). Cuando el servidor envíe la respuesta al *proxy*, éste se la reenviará al navegador. Por los tanto, cuando se utiliza un *proxy* siempre hay dos conexiones TCP: una entre el navegador y el *proxy*, y otra entre el *proxy* y el servidor web, por lo que el *proxy* se encarga de reenviar las peticiones y respuestas entre ellos (tras analizar los contenidos, y cacheándolos si es posible).

El comportamiento de los *proxies* HTTPS es bastante diferente. Aunque las peticiones HTTPS también pueden reenviarse a través de un *proxy*, la sesión TLS se sigue estableciendo entre el navegador y el servidor web, por lo que un *proxy* HTTPS estándar solo puede ver los datos cifrados de la sesión TLS, por lo que no puede analizar o cachear los contenidos intercambiados. Para utilizar un ***proxy* HTTPS**, el navegador establece una conexión TCP con él y le envía una petición especial con el método CONNECT, indicando el nombre de dominio y el puerto TCP del servidor. El *proxy* HTTPS establece una conexión TCP con el destino, pero en este caso no le reenvía la petición CONNECT del cliente, sino que simplemente responde al cliente con una respuesta 200 OK, y a partir de ese momento simplemente reenvía los bytes que reciba del cliente y el servidor entre ellos. De esta forma el cliente puede iniciar la negociación de una sesión TLS con el servidor, incluso si no están conectados directamente sino a través del *proxy*. Una vez establecida la sesión TLS, el cliente y el servidor pueden intercambiarse mensajes HTTP de manera segura. En el **Capítulo 7** estudiaremos técnicas para intentar descifrar y capturar el tráfico de sesiones TLS, aunque esto solo es posible si contamos con la colaboración de uno de los extremos de la comunicación.

A veces, en lugar de emplear un *proxy* cliente (que requiere configurar todos los navegadores y resto de aplicaciones para que lo usen), las organizaciones despliegan en su salida a Internet los conocidos como ***proxies* transparentes**, que actúan como *proxies* cliente, aunque los navegadores no son conscientes de que los están usando. Funcionan de la siguiente forma: los navegadores no tienen ningún *proxy* configurado, así que asumen que pueden comunicarse directamente con el servidor web final, por lo que resuelven su nombre DNS e intentan establecer una conexión TCP contra su IP. Sin embargo, antes de salir a Internet, el *proxy* transparente intercepta esos paquetes (para ello el *proxy* debe estar en la ruta de salida a Internet de la organización) y responde a los mismos como si fuera el servidor destino (i.e. falsificando su dirección IP). Así que la conexión TCP del cliente realmente se establece con el *proxy* transparente, y éste establece otra conexión TCP con el servidor web final (utilizando la IP y el puerto destino que ha enviado el cliente). El navegador envía entonces sus peticiones HTTP al *proxy* (con URLs relativas puesto que piensa que está hablando con el servidor), el *proxy* la reenvía al servidor, y luego hace lo mismo con la respuesta (tras analizar los contenidos y cachearlos como un *proxy* tradicional). Por lo tanto, el comportamiento final es exactamente el mismo que con un *proxy* cliente explícito, pero sin que los navegadores (y a veces los usuarios) sean conscientes que sus comunicaciones están siendo monitorizadas. Las sesiones HTTPS también funcionan a través de un *proxy* transparente, porque este actúa como un *proxy* a nivel TCP, aunque, como en el caso del proxy HTTPS explícito, normalmente solo ve el tráfico cifrado y no puede monitorizar la sesión.

En los últimos años también se ha popularizado el uso de los *proxies* en el lado del servidor, denominados ***proxies* inversos**, que se utilizan principalmente como *firewalls* a nivel de aplicación (i.e. WAF – *Web Application Firewall*), filtrando peticiones maliciosas de los clientes (e.g. evitando ataques de *path transversal*, SQL *injection* o *Cross Site Scripting*), así como para reducir la carga de los servidores web finales, por ejemplo, funcionando como caches de contendidos dinámicos, como terminadores de sesiones TLS, o como balanceadores de carga delante de una granja de servidores web. Normalmente los clientes tampoco son conscientes de la existencia de los *proxies* inversos (a no ser por la cabecera Via), porque típicamente se comportan como si fueran el servidor real (i.e. su dirección IP se publica en el DNS asociada al nombre del servidor). La única diferencia es que normalmente solo son capaces de responder a las peticiones HTTP que les envían los clientes si ya las tienen en la cache. De lo contrario, se las reenvían al servidor web real y esperan su respuesta para reenviársela al cliente. Los *proxies* inversos normalmente sí son capaces de inspeccionar el tráfico HTTPS, porque tienen el certificado digital y la clave privada del servidor por el que se hacen pasar, de forma que son los que terminan la sesión TLS con el cliente, y por tanto pueden analizar todo el tráfico intercambiado.

## 2.11. Ejercicio de repaso de la pila de protocolos TCP/IP

Una vez repasados los protocolos más importantes de la pila de protocolos TCP/IP (aunque este capítulo sea probablemente más útil como referencia que como lectura), sería recomendable repasar los conceptos analizando un fichero de captura (tcp_ip.pcap) con la herramienta 'Wireshark' (que se explicará en detalle en el siguiente capítulo).

En particular se ha capturado el tráfico resultante de ejecutar el comando 'wget' para descargar una el contenido de una página web (**Listado 2.23**). El objetivo de este ejercicio es analizar el tráfico capturado e intentar entender quién y por qué envía cada paquete. Todo el tráfico incluido en el fichero de captura pertenece a la aplicación, y se ha eliminado cualquier paquete espurio para simplificar su análisis.

**Listado 2.23** – Configuración del equipo y ejecución del comando 'wget'

```
$ ifconfig
eth0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
      inet 192.168.1.33  netmask 255.255.255.0  broadcast 192.168.1.255
      inet6 fe80::ec1c:80a4:5f2c:1426  prefixlen 64  scopeid 0x20<link>
      ether 00:15:5d:38:01:04  txqueuelen 1000  (Ethernet)
      RX packets 5465154  bytes 1071721836 (1.0 GB)
      RX errors 0  dropped 32269  overruns 0  frame 0
      TX packets 1936312  bytes 369934714 (369.9 MB)
      TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

lo: flags=73<UP,LOOPBACK,RUNNING>  mtu 65536
      inet 127.0.0.1  netmask 255.0.0.0
      inet6 ::1  prefixlen 128  scopeid 0x10<host>
      loop  txqueuelen 1000  (Bucle local)
      RX packets 1260904  bytes 3032676040 (3.0 GB)
      RX errors 0  dropped 0  overruns 0  frame 0
      TX packets 1260904  bytes 3032676040 (3.0 GB)
      TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

$ route -n
Tabla de rutas IP del núcleo
Destino        Pasarela       Genmask        Indic Métric Ref  Uso Interfaz
0.0.0.0        192.168.1.1    0.0.0.0        UG    100    0      0 eth0
192.168.1.0    0.0.0.0        255.255.255.0  U     100    0      0 eth0

$ resolvectl dns
Global:
Link 2 (eth0): 80.58.61.250 80.58.61.254

$ wget http://www.example.com
--1970-01-01 12:00:00--  http://www.example.com/
Resolviendo www.example.com (www.example.com)... 93.184.216.34,
2606:2800:220:1:248:1893:25c8:1946
Conectando con www.example.com (www.example.com)[93.184.216.34]:80... conectado.
Petición HTTP enviada, esperando respuesta... 200 OK
Longitud: 1256 (1,2K) [text/html]
```

```
Guardando como: “index.html”

index.html                  100%[=============================================>]  1,23K  --.-KB/s
en 0s

1970-01-01 12:00:00 (31,8 MB/s) - “index.html” guardado [1256/1256]
```

# 3. Captura y Análisis de Tráfico de Red

Para analizar el tráfico de una aplicación, primero hay que capturarlo. En este capítulo se analizarán las diferentes estrategias disponibles para capturar tráfico de red, así como las principales herramientas disponibles para realizar dicha captura y análisis posterior de la misma.

## 3.1. Estrategias de Captura de Tráfico de Red

En este apartado analizaremos las diferentes estrategias y técnicas que existen en la literatura para capturar el tráfico de una aplicación. Empezaremos por las técnicas que permiten capturar la mayoría del tráfico, pero que requieren tener más control sobre el sistema a analizar o la infraestructura de red en la que se encuentra, e iremos relajando esos requisitos a cambio de reducir paulatinamente el porcentaje de tráfico que se puede capturar.

### 3.1.1. Captura de tráfico en el propio equipo que ejecuta la aplicación

El lugar ideal para capturar el tráfico de una aplicación es realizando la captura de tráfico en el propio equipo o sistema donde ejecuta la aplicación. Es la única forma de poder analizar todo el tráfico que genera la misma, sobre todo con otras aplicaciones dentro del mismo equipo (e.g. utilizando el interfaz de *loopback* de IP). Además, se pueden utilizar más herramientas de análisis (como 'netstat' o 'strace'), en lugar de estar restringido únicamente al análisis del tráfico. En ese caso, es importante elegir bien los interfaces de los que se desea capturar, o utilizar el interfaz *any* para capturar de todos ellos simultáneamente.

Para analizar aplicaciones cliente-servidor, normalmente es más sencillo capturar el tráfico en el lado cliente, dado que normalmente los servidores son menos accesibles para los analistas, procesan una mayor cantidad de tráfico, y la ejecución de herramientas de captura y análisis puede comprometer su rendimiento o estabilidad.

Un último comentario práctico sobre la captura de tráfico en el propio sistema que lo genera. La mayoría de las tarjetas de red implementan diversas funcionalidades de la pila de protocolos TCP/IP para reducir la carga de la CPU principal. Por ejemplo, es bastante habitual que, además del CRC-32 de Ethernet, el núcleo (*kernel*) del sistema tampoco calcule o compruebe los campos de *checksum* de protocolos como IPv4, TCP o UDP, sino que delegue esta funcionalidad en la tarjeta de red (i.e. *Checksum Offloading*). Esa es la razón por la que en algunas capturas de tráfico puede aparecer un gran número de errores de *checksum* (y una de las razones por las que herramientas de análisis como 'Wireshark' no comprueban por defecto los *checksums* recibidos).

Otro efecto, que puede a llegar a ser más disruptivo respecto a la fiabilidad de las capturas en la misma máquina donde se genera o recibe el tráfico, son los mecanismos de delegación de segmentación (*Large Send Offload* o *TCP/Generic Segmentation Offload*) y de recepción de paquetes grandes (*Large Receive Offload*). Ambos mecanismos tienen el mismo objetivo: reducir el número de transferencias entre la CPU y la tarjeta de red, aunque se aplican en sentidos diferentes del tráfico. Los mecanismos de *Large Send Offload* (LSO) o TCP/*Generic Segmentation Offload* (TSO/GSO) permiten que el sistema operativo envíe a la tarjeta de red paquetes mucho más grandes (e.g. 64KB) de lo que permite la MTU de la red, y que sea la tarjeta de red la que los divida en múltiples segmentos TCP, que ya cumplen con las limitaciones de MTU y MSS negociados en la conexión TCP. Por el contrario, el mecanismo de *Large Receive Offload* (LRO) hace que la tarjeta de red agrupe múltiples segmentos TCP en otro super segmento y lo envíe de una vez a la CPU. Si alguna captura de tráfico contiene paquetes con un tamaño superior a la MTU del medio (teniendo en cuenta los jumbogramas Ethernet) o del MSS de la conexión TCP, probablemente sea un efecto de LSO/TSO/GSO o LSO. Para el análisis de protocolos no es muy relevante, puesto que la captura sigue recibiendo toda la información intercambiada por el equipo, pero no coincide con el tráfico realmente enviado por la red y a veces puede ocultar la temporización real de los mensajes. Aunque antes de desactivarlos para aumentar la fiabilidad de la captura, hay que tener en cuenta el impacto del rendimiento que eso puede tener.

**Listado 3.1** – Capacidades de *offloading* de una tarjeta de red en Linux.

```
$ ethtool -k eth0
Features for eth0:
rx-checksumming: on
tx-checksumming: on
        tx-checksum-ipv4: on
        tx-checksum-ip-generic: off [fixed]
        tx-checksum-ipv6: on
        tx-checksum-fcoe-crc: off [fixed]
        tx-checksum-sctp: off [fixed]
scatter-gather: on
        tx-scatter-gather: on
        tx-scatter-gather-fraglist: off [fixed]
tcp-segmentation-offload: off
        tx-tcp-segmentation: off
        tx-tcp-ecn-segmentation: off [fixed]
        tx-tcp-mangleid-segmentation: off
        tx-tcp6-segmentation: off
generic-segmentation-offload: on
generic-receive-offload: on
large-receive-offload: on
...
```

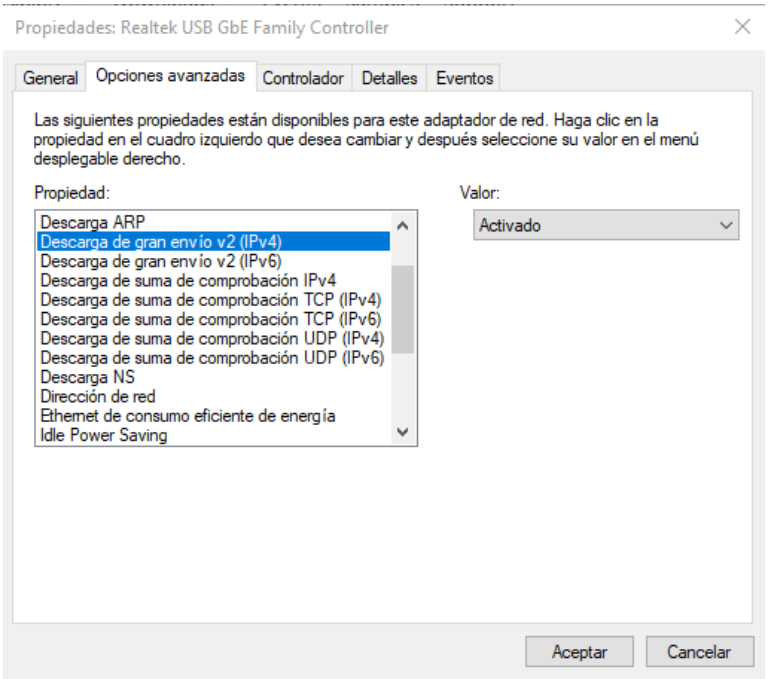



**Figura 3.2** – Capacidades de *offloading* de una tarjeta de red en Windows.

Cada sistema operativo tiene su forma de comprobar y activar/desactivar los mecanismos de *offloading*. Por ejemplo, en Windows se puede acceder consultando las propiedades del interfaz de red (**Figura 3.2**), mientras que en Linux se puede consultar y modificar utilizando el comando 'ethtool' (**Listado 3.1**).

### 3.1.2. Captura de tráfico a nivel físico o de enlace

Si el sistema que ejecuta la aplicación no es accesible para el analista, o no dispone de las herramientas de captura o análisis necesarias (o los permisos para ejecutarlas), el siguiente mejor punto para capturar el tráfico es con una **sonda de red** física (*network tap* en inglés) a la que se conecta el equipo y que replica todos los paquetes que envía y recibe el mismo a otros interfaces de red a los que se conecta la estación de captura.

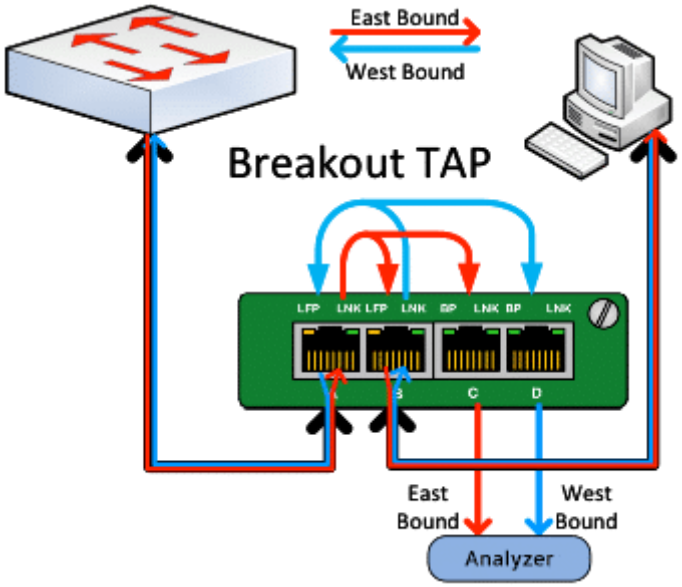



**Figura 3.3** – Esquema de una sonda de red (*network tap*) [NeoxNetworks]

La **Figura 3.3** muestra el esquema de una sonda de red que captura el tráfico entre un equipo con una interfaz de red Ethernet y su *switch*. El *switch* se conecta al interfaz A, el equipo en el interfaz B, y la estación de captura se conecta a los interfaces C y D. La sonda se encarga de reenviar todos los paquetes que se envían por los interfaces A y B entre ellos como si siguiesen conectados a través de un cable, pero también envía una copia de cada paquete que se recibe por los interfaces A y B a la estación de captura a través de los interfaces C y D (que normalmente solo pueden enviar tráfico, pero no recibir). Se necesitan dos interfaces de salida con la misma tasa que los entrantes para garantizar que se pueden recibir todos los paquetes (o un puerto de más capacidad). Esto es, un puerto Gigabit Ethernet es capaz de enviar y recibir tramas a 1 Gbps en cada sentido. Por lo tanto, si el equipo envía a 1 Gbps y recibe datos del switch también a 1 Gbps, se necesitan enviar 1+1=2 Gbps a la estación de captura.


**Figura 3.4** – A) Sonda de red activa [Garland]. B) Sonda de red pasiva. [GreatScottGadgets]

En interfaces de fibra óptica (donde se puede desviar parte de la luz a la estación de captura) o de baja velocidad (hasta 100 Mbps y a distancias cortas), se puede utilizar una sonda pasiva (**Figura 3.4.B**), que no necesita alimentación porque simplemente interconecta los cables de recepción del puerto A con los cables de transmisión de los puertos B y C, y los cables de recepción del puerto B con los cables de transmisión de B y D.

Sin embargo, para enlaces Ethernet de cobre a 1 Gbps o más se necesita una sonda activa (**Figura 3.4.A**), que reciba las tramas de los equipos bajo estudio y envíe una copia de las mismas a la estación de captura. Además, algunas sondas activas permiten agregar los dos sentidos de la comunicación en un único puerto de mayor capacidad. La pega de las sondas activas, además de que son mucho más caras que las pasivas, es que no funcionan sin alimentación, aunque suelen implementar un mecanismo para puentear físicamente los interfaces A y B cuando se pierde la alimentación.

Para capturar el tráfico de más de un equipo de la LAN también se puede usar una sonda física en el enlace de *uplink* del *switch* de acceso (al que se conectan los equipos finales) con el de distribución (al que se conectan los *switches* de acceso), aunque en ese caso ya no se podría capturar el tráfico enviado directamente entre equipos en el mismo *switch* de acceso (al reenviarse localmente y no pasar por el *uplink*). Pero esta estrategia requeriría instalar una sonda de red y probablemente una estación de captura por cada *switch* de acceso que se quisiese monitorizar, por lo que no es muy escalable. En ese caso es mucho mejor utilizar las capacidades de *port mirroring* del *switch* (si dispone de ella).

***Port mirroring*** es una funcionalidad avanzada que tienen muchos *switches* gestionados para replicar el comportamiento de una sonda de red física, y que permite copiar todo el tráfico enviado y recibido por uno o más puertos, a otro puerto del mismo *switch* donde se conecta la estación de captura. Por ejemplo, en los *switches* Cisco a esta funcionalidad se le denomina SPAN (ver **Figura 3.5**), y permite enviar el tráfico copiado incluso a otro *switch* remoto con RSPAN, o encapsulado en un túnel IPv4-GRE a una estación de captura remota con ERSPAN.

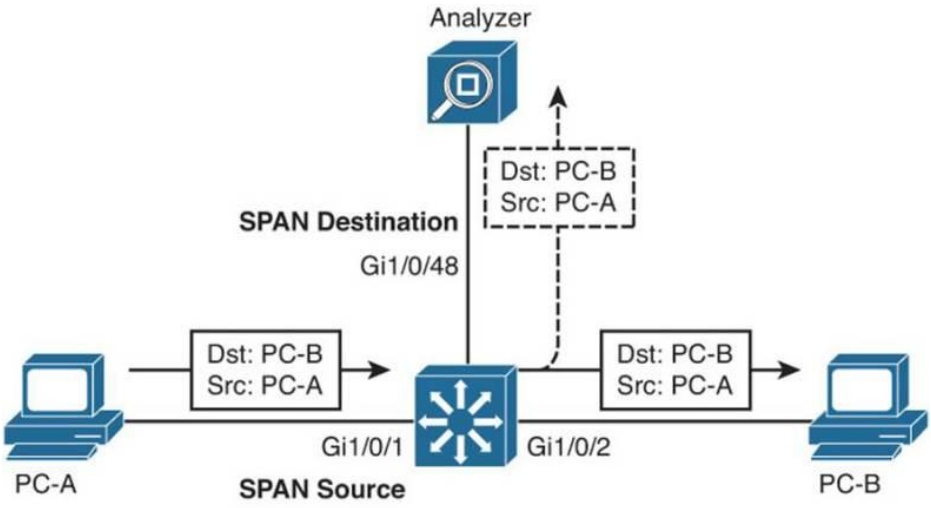



**Figura 3.5** – Ejemplo de *port mirroring* con Cisco SPAN [imd.guru]

La principal ventaja de la técnica de *port mirroring* es la flexibilidad: además de poder realizar la copia de tráfico de manera lógica en lugar de con una instalación física, los switches suelen permitir capturar el tráfico de múltiples interfaces simultáneamente, eligiendo qué VLANs y qué dirección de tráfico se quiere copiar, y enviar todo el tráfico a un único puerto de salida, que puede tener una tasa diferente a los puertos siendo monitorizados (e.g. se puede agrupar el tráfico de varios puertos 1GbE en un puerto de 10GbE de salida).

La desventaja del *port mirroring* frente a las sondas de red es la fiabilidad de la captura. Las sondas de red garantizan que la estación de captura recibe una copia de cada trama que atraviesa en el enlace monitorizado en el momento que se recibe. Sin embargo, con *port mirroring* el *switch* debe tener capacidad suficiente para realizar la copia de las tramas, encolarlas en el puerto de salida y enviarlas cuando sea posible a la estación de captura. Cuantos más puertos y más tráfico se agrupen en un único puerto de salida, más probable es que se pierdan tramas y que la estación de captura las reciba más tarde o desordenadas.

Por lo tanto, aunque el *port mirroring* es muy útil para capturar el tráfico de un gran número de puertos de manera rápida y flexible por ejemplo para monitorizar el tráfico con un IDS (*Intrusion Detection System*), las sondas de red son preferibles para obtener una copia fidedigna del tráfico de una estación.

En cuanto a la configuración de la estación de captura, tanto si se utiliza *port mirroring* como si se utiliza una sonda de red, es muy importante que los interfaces de red Ethernet que capturen el tráfico se encuentren en modo promiscuo, puesto que la gran mayoría de las tramas que reciba (excepto las tramas *broadcast*/*multicast*) no estarán destinadas a su dirección MAC. Otra precaución habitual es no configurar una dirección IP en los interfaces de captura, de forma que la estación de captura no pueda enviar tráfico a través dichos interfaces, así como

desactivar el reenvío de paquetes IP (i.e. para que la estación de captura no se comporte como un *router*).

Y por último una advertencia sobre cómo NO se debería capturar tráfico. Si se consulta documentación antigua, se puede encontrar como recomendación utilizar un ***hub* Ethernet** para capturar tráfico, puesto que los *hubs* son dispositivos de nivel físico que reenvían as tramas por todos los puertos, excepto por el que llegó (lo mismo que hace un *switch* Ethernet con las direcciones *broadcast*, *multicast* o MAC desconocidas). Por lo tanto, actúa como una especie de sonda de red, porque todos los puertos ven todo el tráfico. De hecho, este era el comportamiento de las redes Ethernet originales, donde para ver todo el tráfico de la LAN bastaba con poner el interfaz de red de cualquier estación en modo promiscuo (por eso esta operación suele requerir permisos de superusuario). Sin embargo, ya no se recomienda esta práctica porque los *hubs* degradan significativamente el rendimiento de Ethernet, puesto que convierten el enlace en semidúplex (i.e. solo es posible enviar o recibir una trama), y si varias estaciones transmiten a la vez se produce una colisión y hay que retransmitir las tramas afectadas (esa es la razón por la que las tramas Ethernet tienen un tamaño mínimo de 64 octetos), además la velocidad máxima de los *hubs* nunca pasó de 100 Mbps.

Otra alternativa, aún menos recomendable, es atacar el *switch* de acceso para intentar que se comporte como un *hub*, saturando su tabla de reenvío (que suele estar implementada en *hardware* y tiene una capacidad limitada) con direcciones MAC aleatorias, de forma que éstas sobrescriban las entradas de las direcciones MAC reales, y por tanto el *switch* no sepa cómo encaminar las tramas y las envíen por todos los puertos. Sin embargo, esta es una técnica muy poco fiable, puesto que requiere bombardear continuamente el *switch* con direcciones MAC, dado que en cuanto vea una trama legítima se acordará durante un tiempo de la MAC origen, y le puede reenviar algo de tráfico hasta que se sobrescriba con una falsa. Además, los *switches* modernos tienen mecanismos de defensa frente a este ataque (i.e. limitando la cantidad de direcciones MAC que se pueden aprender de un puerto de acceso). Y sobre todo porque este tipo de ataques puede degradar severamente el rendimiento, no solo del *switch* atacado, sino de toda la LAN, ya que se dispara la cantidad de tráfico desconocido en la misma.

Otro ataque contra el *switch*, denominado Robo de Puerto (*Port Stealing* en inglés), consiste en intentar envenenar la tabla de reenvío del *switch*, enviando desde la estación de captura tramas con la dirección MAC de la víctima, de forma que el *switch* piense que se ha cambiado de puerto y comience a enviarnos las tramas destinadas al mismo a nosotros. Este ataque es menos peligroso para la estabilidad del *switch* y de la LAN que el anterior, pero tiene el problema que en cuanto la víctima envía una trama, la tabla de reenvío del *switch* vuelve a cambiar y le pueden llegar algunas tramas directamente, por lo que hay que estar inyectando tramas

falsas continuamente. De hecho, para conseguir que el tráfico le llegue finalmente a la víctima, hay que conseguir que esta envíe alguna trama (se pueden utilizar peticiones ARP para ello) para restaurar el puerto en el *switch*, enviar el tráfico y volver a engañar al *switch* para que nos reenvíe la siguiente trama. Además, solo permite captura el tráfico destinado hacia la víctima, así que, si también se quiere capturar el tráfico saliente, también es necesario "robar" el puerto del *router* por defecto, lo que es mucho más disruptivo, ya que desvía a la estación de captura el tráfico saliente de todas las estaciones de la subred y además el *router* suele enviar mucho más tráfico, por lo que hay más probabilidad de que el *switch* aprenda el puerto correcto y se pierda tráfico. En general, los ataques al reenvío de los *switches* son muy ruidosos y poco fiables, así que no son nada recomendables.

### 3.1.3. Captura de tráfico a nivel de red

Si por alguna razón no es posible tener acceso al sistema donde ejecuta la aplicación a analizar, ni siquiera de forma física para instalar una sonda de red, ni la capacidad de configurar su *switch* de acceso (o éste no dispone de capacidades de *port mirroring*). La siguiente alternativa es intentar capturar su tráfico a nivel de red. Para ello existen diferentes técnicas, así que empezaremos describiendo las más robustas, hasta llegar a las que dependen de ataques activos. En cualquier caso, todas ellas tienen la limitación de que no son capaces de capturar todo el tráfico del equipo siendo monitorizado, porque típicamente no pueden obtener el tráfico que el objetivo envía a otro equipo dentro de la misma subred, solo el tráfico que sale de la subred (aunque en aplicaciones cliente-servidor esto no debería ser un problema, a no ser que el servidor se encuentre en la misma subred que el cliente).

Y es que el objetivo fundamental de este tipo de ataques es conseguir que el tráfico IP del objetivo pase por la estación de captura, típicamente convirtiendo a ésta en su *router* por defecto, aunque para ello la estación de captura debe estar en la misma subred que el objetivo. La forma más directa sería **modificando la tabla de rutas del equipo**, cambiando el siguiente salto de la ruta por defecto (0.0.0.0/0) por la dirección IP de la estación de captura, aunque para ello típicamente se necesitaran permisos de superusuario, lo que es poco probable si ya estamos evaluando estas alternativas.

Hay formas de modificar la tabla de rutas de un sistema desde el exterior, siempre y cuando esta no haya sido configurada de manera estática, y es que la mayor parte de los equipos finales obtienen su dirección IP, servidor DNS y *router* por defecto de manera dinámica utilizando mecanismos como DHCP, en el caso de IPv4, y SLACC (*Stateless Address Autoconfiguration*) o DHCPv6, en el caso de IPv6. Así que, si disponemos acceso a la configuración del *router* (en cuyo caso se debería comprobar si tiene capacidades de *port mirroring*, algunos *routers* también disponen

de ella) o del servidor DHCP de la subred, bastaría con configurarlo para que el equipo objetivo (o todos los equipos de la subred, pero en ese caso habría que reenviar todo el tráfico y capturar únicamente el del sistema objetivo) tenga como *router* por defecto a la estación de captura (Opción DHCP 3).

Si tampoco tenemos control sobre la infraestructura de red, la única alternativa consiste en la realización de **ataques activos**, lo que obviamente debe considerarse únicamente como última opción, y teniendo el consentimiento expreso de los responsables de la infraestructura antes de realizarlo. Una de las herramientas clásicas para realizar este tipo de ataques es Ettercap [Ettercap] (ver **Figura 3.6**), que habría que ejecutar en la máquina de captura para realizar un ataque de *Adversary-in-the-Middle* (AitM) de forma que todo el tráfico IP pase por ella.

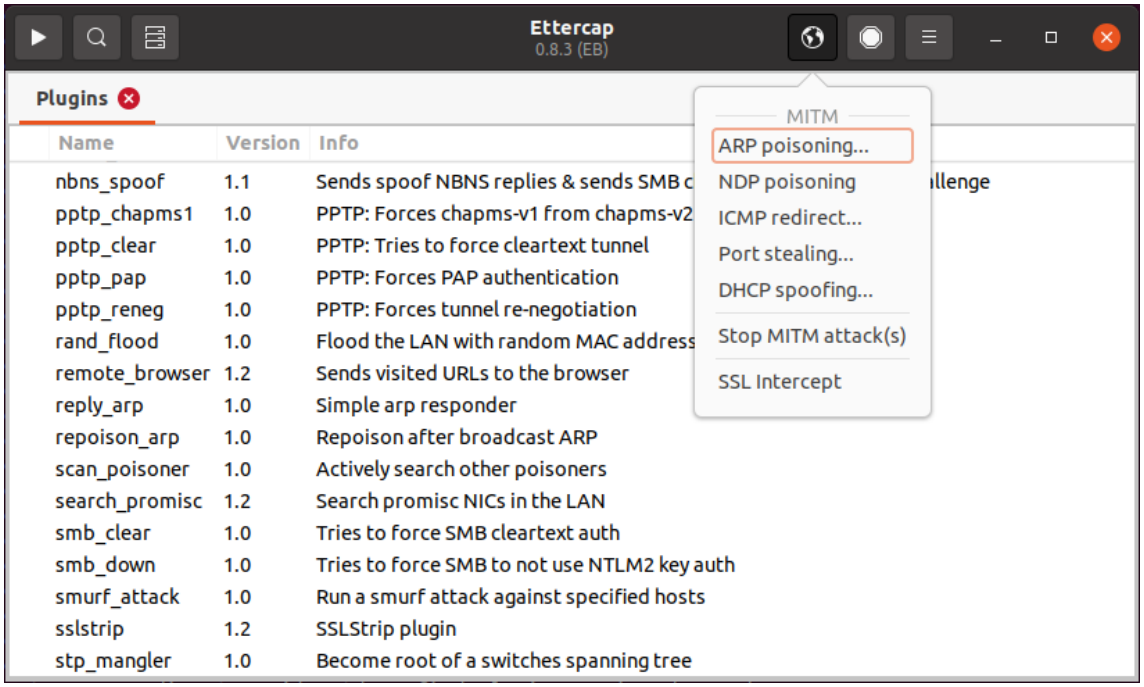



**Figura 3.6** – Ataques de *Man-in-the-Middle* (MitM) y *plugins* de 'Ettercap' [Ettercap]

El **ataque de redirección ICMP** consiste simplemente en enviar a la víctima mensajes de ICMP *Redirect* (ver apartado sobre ICMP en el Capítulo 2) haciéndose pasar por el *router* por defecto de la subred, indicando que el *router* más adecuado para todos los datagramas de la víctima es el atacante. Aunque cada vez quedan menos sistemas que aceptan mensajes de ICMP *Redirect* por defecto, así que la efectividad de este ataque puede ser limitada.

El **ataque de suplantación DHCP** (DHCP *Spoofing*, en inglés) consiste en crear un servidor DHCP ficticio que se queda a la espera de que la víctima solicite una dirección IP, y en ese momento intenta responder antes que el servidor DHCP legítimo, proporcionando una configuración que permita realizar un ataque de *Adversary-in-the-Middle* (AitM), típicamente cambiando el *router* por defecto a la

dirección IP del atacante. Además, nos permite establecer la dirección IP del sistema, lo que simplifica la identificación y captura de su tráfico. Este ataque depende mucho de la velocidad de respuesta de la infraestructura de red, aunque suele ser efectivo en redes donde el servidor DHCP legítimo no está en la misma subred, sino que se emplea un *relay* DHCP para enviar las peticiones a un servidor DHCP centralizado (lo cual es muy habitual en redes empresariales), y por tanto tarda más en responder.

Si no es posible modificar la tabla de rutas del sistema a monitorizar, se necesita a pasar a ataques más agresivos, como el **envenenamiento** (*poisoning*, en inglés) **de la cache ARP/*Neighbor Discovery* (ND)**. Este ataque intenta modificar la caché ARP de la víctima, de forma que asocie la dirección IP del *router* por defecto (que no podemos cambiar) con la dirección MAC del atacante, de forma que la víctima nos reenvíe sus datagramas, pensando que se las está enviando al router. El problema de este ataque es que, para capturar el tráfico en ambas direcciones, también hay que atacar la cache ARP del *router*, de forma que cuando éste quiera mandar tráfico al sistema víctima éste también pase por nosotros. Hay varias formas de implementar este ataque, como intentar responder a las consultas ARP antes que la víctima (como en el ataque de suplantación DHCP), aunque Ettercap implementa un mecanismo más robusto enviando continuamente mensajes ARP no solicitados a la dirección MAC de la víctima y del *router*, asociando su dirección MAC a la dirección IP del otro. Por lo tanto, es un ataque mucho más sencillo que el robo de puerto (*Port Stealing*) explicado en el apartado anterior.

Es importante conocer cómo se implementan estos ataques para de esta forma poder interpretar el tráfico capturado, porque éste suele contaminarse con el tráfico generado por Ettercap, sobre todo cuando se utilizan los ataques más agresivos.

Una limitación muy importante de todas las técnicas que hemos visto hasta ahora (tanto a nivel físico, de enlace o de red) consiste en que la estación de captura tiene que estar conectada al equipo, al *switch* o a la subred donde se encuentra el equipo. Cuando la estación de captura no se puede situar tan cerca del equipo a analizar, hay tres alternativas:

- Si se tienen permisos de superusuario en el equipo, y este lo soporta, se podría establecer un **túnel IP o GRE** (i.e. encapsulando paquetes IP dentro de otros paquetes IP) o una VPN (e.g. IPSec, SSL, L2TP) con un terminador que se encarga de desencapsular el tráfico, capturar una copia y reenviarlo. En este caso el terminador puede estar en cualquier localización, con tal de que sea accesible vía IP por el equipo, aunque hay que tener cuidado con el direccionamiento que se proporciona al equipo tunelizado, para que pueda seguir comunicándose con el servidor de aplicación.

- Capturar o desviar el tráfico desde el *router* por defecto de la subred que utiliza el sistema. Obviamente esto depende en gran medida de las capacidades de dicho *router* y el control que tengamos sobre el mismo, pero en *routers* comerciales se pueden utilizar mecanismos de *policy routing* para reenviar el tráfico de un origen específico de una manera personalizada o, si se conoce el destino del tráfico de la aplicación, simplemente estableciendo una ruta estática, donde el siguiente salto hacia él es la estación de captura (aunque hay que tener cuidado si la dirección IP del destino cambia). Con ambos mecanismos la estación de captura debe estar directamente conectada al *router*, de lo contrario sería necesario establecer un túnel IP/GRE con ella, tal como se menciona en el punto anterior. En los *routers* basados en Linux (e.g. OpenWRT), se pueden implementar las dos técnicas, tal como muestran los **Listados 3.7** (ruta estática) y **3.8** (*policy routing*).

- Capturar el tráfico a nivel de aplicación, si el protocolo en cuestión permite el uso de *proxies*. Tal y como se describe en el siguiente apartado.

**Listado 3.7**: Configuración de una ruta estática en un *router* Linux.

```
router# export IP_SERVIDOR_DESTINO=93.184.216.24
router# export IP_ESTACION_CAPTURA=192.168.2.100
router# ip route add ${IP_SERVIDOR_DESTINO}/32 via ${IP_ESTACION_CAPTURA}
router# ip route
default via 192.168.2.2 dev eth1
192.168.1.0/24 dev eth0 proto kernel scope link src 192.168.1.1 metric 100
192.168.2.0/24 dev eth1 proto kernel scope link src 192.168.2.1 metric 100
93.184.216.24 via 192.168.2.100 dev eth1
```

**Listado 3.8**: Configuración de *policy routing* en un *router* Linux.

```
router# export IP_SISTEMA_ORIGEN=192.168.1.33
router# export IP_ESTACION_CAPTURA=192.168.2.100
router# ip route add default via ${IP_ESTACION_CAPTURA} table to_capture_table
router# ip route list table to_capture_table
default via 192.168.2.200 dev eth0
router# ip rule add from ${IP_SISTEMA_ORIGEN} lookup to_capture_table
router# ip rule list
0:      from all lookup local
32765:  from 192.168.1.33 lookup to_capture_table
32766:  from all lookup main
32767:  from all lookup default
```

En cuanto a la estación de captura, su configuración cuando se captura tráfico a nivel de red es totalmente opuesta a la empleada en la captura de tráfico a nivel de enlace. Para empezar, no suele ser necesario poner el interfaz de red en modo promiscuo, puesto que todo el tráfico suele estar destinado a la estación de captura (al fin y al cabo, se comporta como el *router* por defecto o realiza un ataque de AitM). Por lo tanto, también suele necesitar tener una dirección IP configurada en el interfaz de captura y, si se desea que el sistema a analizar tenga conectividad con el exterior de la subred, es necesario activar el reenvío de datagramas entre interfaces de red (i.e. debe comportarse como un *router*). Por último, es recomendable capturar tráfico únicamente en el interfaz que da al sistema siendo monitorizado, porque de lo contrario los paquetes aparecerán dos veces, una cuando se reciben y otra cuando se reenvían.

### 3.1.4. Captura de tráfico a nivel de aplicación

La captura de tráfico a nivel físico, de enlace, o de red es la mejor forma de maximizar la cantidad de tráfico capturada para así poder identificar el protocolo o protocolos que emplea la aplicación a analizar. Sin embargo, a veces sí se conocen a priori ciertos detalles sobre la misma que nos permiten saber algo de los protocolos empleados. Por ejemplo, hay un gran número de aplicaciones, incluido el *malware*, que encapsulan su tráfico dentro mensajes HTTP para así pasar más desapercibidos, y fundamentalmente para poder funcionar en redes empresariales que no permiten la conexión directa de los equipos con Internet. En esos casos, o cuando se utiliza cifrado con TLS, puede tener sentido capturar el tráfico directamente a nivel de aplicación.

Sin embargo, esto solo suele ser posible en protocolos que permiten el reenvío de mensajes a nivel de aplicación. Por suerte la mayoría de los protocolos más populares de Internet tienen esta capacidad, incluyendo: DNS, HTTP o SMTP. Lo que realmente define la capacidad de una aplicación pueda utilizar un *proxy* o un servidor intermedio, es que no dependa únicamente de la dirección IP y puerto destino para identificar cuál es su destino, sino que el nivel de aplicación contenga información sobre el destinatario del mensaje (al menos en las peticiones, porque el *proxy* puede asociar las respuestas con las peticiones). Por ejemplo, en DNS el campo consulta incluye en nombre DNS completo por el que se pregunta, por lo que cualquier servidor recursivo sabe a qué servidores de nombres debería preguntar. Los mensajes SMTP por su parte incluyen las direcciones de correo de los destinatarios (e.g. alice@example.com), que siempre incluyen un dominio DNS, por lo que cualquier servidor de correo sabe a qué servidor de correo debe entregarle dicho correo (basta con preguntar por el registro DNS de tipo MX del dominio, que devuelve el nombre del servidor de correo entrante de la organización). Por último, cuando un navegador utiliza un *proxy* HTTP envía la URL completa, por lo que el *proxy* puede saber el servidor web con el que debe contactar

(además, a partir de HTTP/1.1 la cabecera obligatoria Host también incluye esa información). Por lo tanto, si la aplicación utiliza un protocolo que permite su paso por *proxies*, se puede intentar desviar su tráfico a un *proxy* que esté bajo nuestro control. Para ello, lo más sencillo es configurar la aplicación para que lo utilice directamente. Por ejemplo, en una aplicación de correo, configurándola para que su servidor SMTP de salida sea nuestro *proxy*.

Obviamente la forma de configurar esto depende de la propia aplicación, aunque muchos sistemas operativos, como Windows (**Figura 3.9.A**), permiten configurar el *proxy* HTTP de salida de manera global, puesto que la mayoría de las aplicaciones emplean esa configuración (e.g. Chrome), aunque algunas también permiten su propio *proxy* manualmente (e.g. Firefox – **Figura 3.9.B**), lo que tiene menos impacto en la configuración global del sistema.

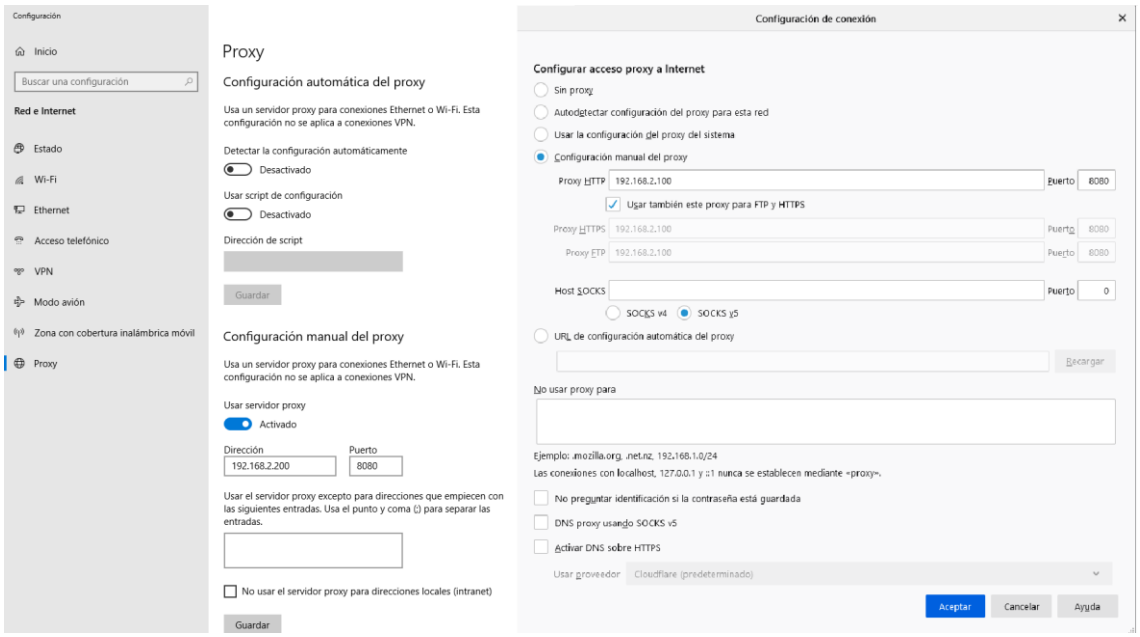



**Figura 3.9** - A) Configuración de *proxy* en Windows 10 y B) Firefox

En los sistemas operativos Linux/UNIX, muchas aplicaciones consultan dos variables de entorno denominadas 'http_proxy' y 'https_proxy', que definen la URL del proxy HTTP y HTTPS que debe emplear la aplicación (e.g. "export http_proxy=http://192.168.2.100:8080"), aunque algunas aplicaciones no las soportan y hay que pasarlas como un parámetro (e.g. la máquina virtual Java define las opciones "-Dhttp.proxyHost=192.168.2.100 -Dhttp.proxyPort= 8080" y "-Dhttps.proxyHost=192.168.2.100 -Dhttps.proxyPort=8081").

Algunas aplicaciones también tienen la capacidad de utilizar *proxies* SOCKS [RFC1928], que es un protocolo agostico del nivel de aplicación, que permite a protocolos basados en TCP o UDP enviar el tráfico a un *proxy* SOCKS dentro de la organización (indicando cual es la dirección IP y el puerto del destino), que es el que

luego reenvía los datos a destino final, o incluso recibir conexiones externas a través del *proxy* SOCKS.

Para analizar las aplicaciones basadas en HTTP u otro protocolo basado en TCP que soporte *proxies* utilizaremos la herramienta 'mitmproxy', explicada en el **Apartado 3.2.3**.

Si el protocolo no soporta *proxies*, ni de manera nativa ni mediante protocolos como SOCKS (o no podemos establecernos), y no se puede capturar su tráfico ni cambiar su destino, la única alternativa que queda es intentar redirigir el tráfico de la aplicación hacia un *proxy* inverso que luego envíe todo el tráfico al destino real (para ello también podemos usar 'mitmproxy'). Para ello existen dos alternativas: redirigir el tráfico al *proxy* inverso utilizando un NAT, o alterar el DNS para que cuando pregunte por el nombre de dominio del destino devolvamos la dirección IP del *proxy* inverso.

Aunque en la mayoría de caso el NAT se utiliza para cambiar la dirección y el puerto origen de los datagramas, para así traducir entre direcciones privadas y públicas (y viceversa). Muchos *routers* también tienen la capacidad de cambiar la dirección y puerto destino de los paquetes que los atraviesan, con una funcionalidad denominada **NAT Destino (DNAT)**. El **Listado 3.9** muestra la configuración de NAT Destino de un router basado en Linux (e.g. 'OpenWRT'), aunque obviamente cada familia de *routers* puede tener su propia forma de configurar esta funcionalidad. Una ventaja de esta técnica es que el *proxy* inverso ya no tiene que estar directamente conectado al *router* que encamina el tráfico de la aplicación.

**Listado 3.9** – Configuración de NAT Destino en un *router* Linux para redirigir tráfico HTTP.

```
router# export IP_SISTEMA_ORIGEN=192.168.1.33

router# export IP_SERVIDOR_DESTINO=93.184.216.24

router# export IP_ESTACION_CAPTURA=192.168.4.100

router# iptables -t nat -A PREROUTING -p tcp -s ${IP_SISTEMA_ORIGEN} -d
${IP_SERVIDOR_DESTINO} --dport 80 -j DNAT --to-destination ${IP_ESTACION_CAPTURA}

router# iptables -t nat -n -L PREROUTING
Chain PREROUTING (policy ACCEPT)
target   prot opt  source      destination
DNAT     tcp  --   192.168.1.33 93.184.216.24    tcp dpt:80 to:192.168.4.100
```

Si no tenemos control sobre alguno de los routers por los que pasa el tráfico, la otra alternativa consiste en interceptar el tráfico DNS de la aplicación. Para ello se requiere: acceso al sistema origen, acceso a la infraestructura DNS que emplea el

sistema origen, o la capacidad de desviar el tráfico DNS del sistema origen a nuestro propio servidor DNS.

Si se tiene acceso al sistema cliente, el cambio más sencillo consiste en añadir el destino al servidor HOSTS del sistema (i.e. C:\Windows\System32\drivers\etc\hosts en Windows, o /etc/hosts en Linux/UNIX) apuntando a nuestra dirección IP. Pero eso normalmente requiere permisos de superusuario. La otra alternativa consiste en cambiar los servidores recursivos que utiliza el sistema para que utilice el nuestro, aunque obviamente este tiene la desventaja de que el cambio afecta a todas las aplicaciones del sistema. En los sistemas UNIX la dirección IP de los servidores recursivos se configuran en /etc/resolv.conf, o usando el comando 'resolvectl' en los sistemas Linux que utilizan 'systemd', y en Windows se configura en las propiedades TCP/IPv4 del interfaz de red (**Figura 3.10**), aunque esto también suele requerir permisos de administrador. Aunque a veces los paneles de configuración gráfica sí permiten cambiar esa información a usuarios no administradores (**Figura 3.11**).

Si no se tiene acceso al sistema, se podría intentar modificar mediante el servicio DHCP (Opción 6), si es que el sistema origen utiliza DHCP para obtener esta información. Para ello es necesario tener control sobre la infraestructura DHCP existente, o mediante un ataque de suplantación DHCP explicado en el apartado anterior.

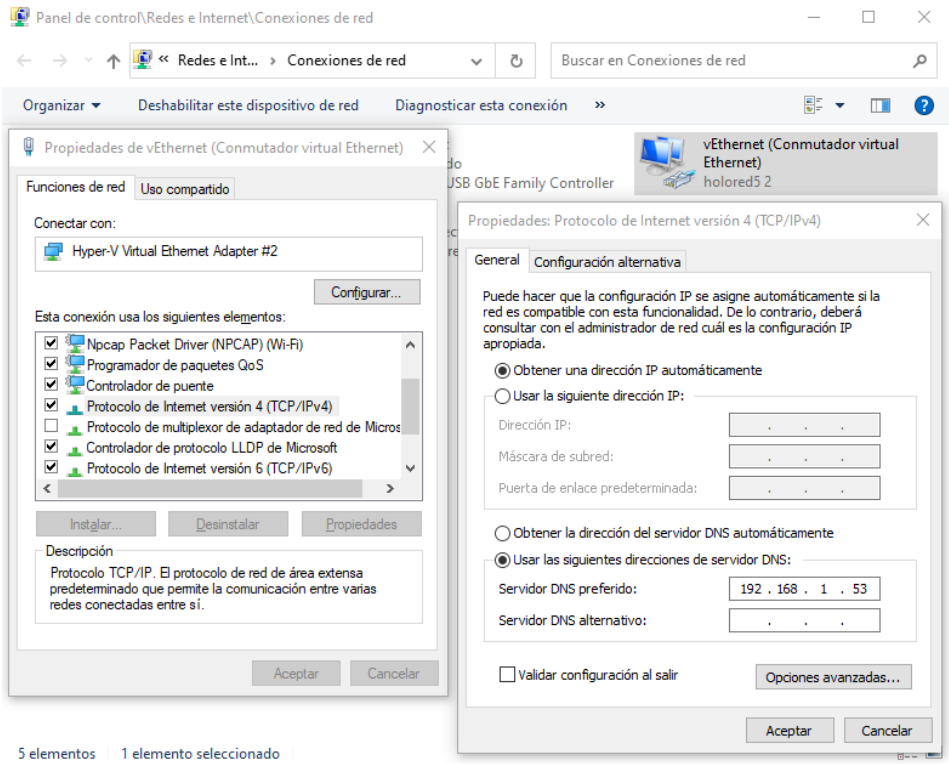



**Figura 3.10** – Configuración DNS de Windows.


**Figura 3.11** – Panel configuración de propiedades de red de Ubuntu Linux.

La última alternativa para redirigir el tráfico DNS a nuestro servidor es utilizando la técnica de NAT Destino (**Listado 3.12**). En ese caso se podría redirigir el tráfico de la aplicación directamente al *proxy*, pero si también controlamos el tráfico DNS, podemos tener control total sobre la IP del servidor destino.

**Listado 3.12** – Configuración de NAT Destino en un *router* Linux para redirigir tráfico DNS.

```
router# export IP_SISTEMA_ORIGEN=192.168.1.33

router# export IP_SERVIDOR_DNS=80.58.61.250

router# export IP_ESTACION_CAPTURA=192.168.1.53

router# iptables -t nat -A PREROUTING -p udp -s ${IP_SISTEMA_ORIGEN} -d ${IP_SERVIDOR_DNS}
--dport 53  -j DNAT --to-destination ${IP_ESTACION_CAPTURA}

router# iptables -t nat -n -L PREROUTING
Chain PREROUTING (policy ACCEPT)
target   prot opt  source       destination
DNAT     udp  --   192.168.1.33   80.58.61.250    udp dpt:53 to:192.168.1.53
```

Si conseguimos redirigir el tráfico DNS a una dirección bajo nuestro control con cualquiera de las técnicas anteriores, podemos instalar un servidor DNS recursivo en dicha IP, pero que además tenga la capacidad de trabajar como un servidor autoritativo para una zona DNS (i.e. para poder cambiar la dirección IP del servidor destino). Una buena alternativa podría ser 'dnsmasq' [dnsmasq], que es un conjunto de demonios para proporcionar servicios de DHCP (lo que además nos puede

permitir cambiar la configuración DNS del sistema vía DHCP) y de DNS. La configuración de 'dnsmasq' (/etc/dnsmasq.conf) es relativamente sencilla. Por defecto solo se activa el servidor DNS (el servidor DHCP hay que activarlo explícitamente), y tiene la característica que el fichero /etc/hosts se consulta antes que los servidores recursivos. Por lo que bastaría con añadir al destino al fichero /etc/hosts, para cambiar dirección IP. Aunque es importante que 'dnsmasq' también sea capaz de resolver el resto de direcciones IP, por lo que también hay que configurar al menos un servidor DNS recursivo al que enviar el resto de peticiones. Por defecto 'dnsmasq' utiliza los servidores configurados en /etc/resolv.conf.

Una vez controlemos la resolución DNS del sistema origen, podemos analizar las consultas DNS que realiza la aplicación, y hacer que el nombre DNS del servidor destino se resuelva con la dirección IP del *proxy* inverso, como por ejemplo la herramienta 'mitmproxy' que describiremos en el **Apartado 3.2.3**.

Por último, cuando la aplicación a analizar no tiene ningún cliente y se basa en HTTP, es posible analizarla directamente desde el navegador, puesto que los navegadores más populares disponen de un panel para desarrolladores web (al que se puede acceder pulsando F12), que también permite analizar el comportamiento de red, mostrando todos los mensajes HTTP enviados y recibidos por el navegador (**Figura 3.13**).

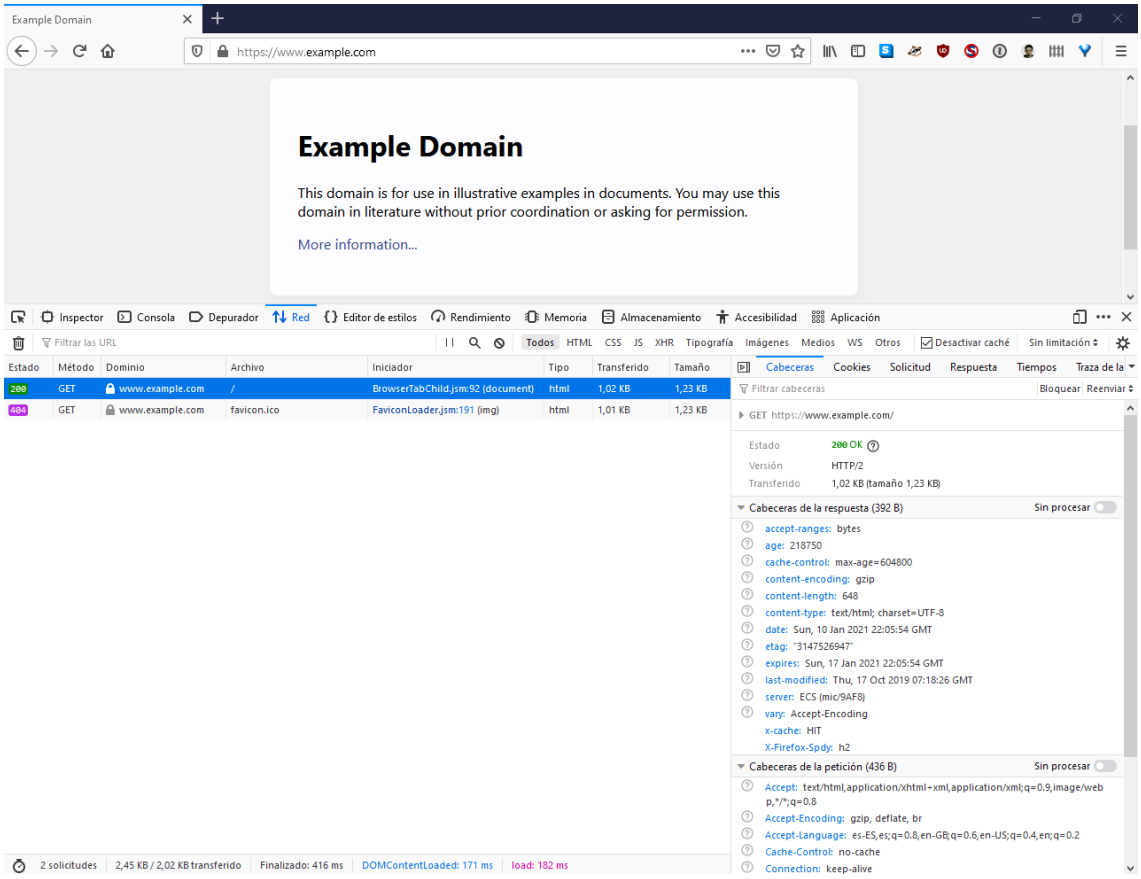



**Figura 3.13** – Panel de desarrollo web de Firefox.

## 3.2. Herramientas de Captura y Análisis de Tráfico

Aunque existen un gran número de herramientas comerciales para capturar y analizar tráfico tanto a nivel de red como a nivel de aplicación, en este curso vamos a centrarnos en las tres herramientas de código abierto más populares: 'tcpdump', 'wireshark' y 'mitmproxy'.

### 3.2.1. Captura de tráfico con 'tcpdump'

'tcpdump' [tcpdump] y su librería asociada libpcap, son las herramientas de referencia para la captura de tráfico TCP/IP. Ambos proyectos son multiplataformas y se ha portado a casi cualquier sistema operativo existente, incluyendo Windows, Linux, macOS y demás UNIX.

'tcpdump' es una herramienta de línea de comandos que permite capturar tráfico en tiempo real y mostrar un resumen de cada paquete en modo texto. Esto puede ser útil para comprobar rápidamente si se está capturando tráfico en un interfaz y el tipo de tráfico, pero normalmente no proporciona suficiente detalle como para

realizar un análisis en profundidad del tráfico que genera una aplicación, así que en esta descripción de uso obviaremos la gran cantidad de opciones que controlan la información que se representa [man], y nos centraremos en el uso de 'tcpdump' para capturar tráfico y almacenarlo en un fichero, que normalmente tiene la extensión .pcap.

Los ficheros pcap tienen contenido binario y una estructura muy sencilla: empiezan con una cabecera global que incluye el número mágico (0xa1b2c3d4), la fecha en la que se realizó la captura, la longitud máxima de los paquetes capturados, el tipo de enlace, y a continuación la lista de paquetes capturados que, además del paquete en sí, incluye un *timestamp* de cuándo se capturó, la longitud capturada y la longitud real que tenía el paquete. Aunque los ficheros pcap están soportados por cualquier herramienta de análisis de tráfico que se precie, 'Wireshark' utiliza por defecto el formato pcapng (PCAP *Next Generation*) que, aunque es compatible con el formato original, permite almacenar, además de los paquetes capturados, información adicional sobre los mismos, como por ejemplo comentarios de los paquetes, información sobre el interfaz de captura, información sobre la resolución de nombres, etc. La librería libpcap permite acceder a los contenidos de los ficheros pcap, aunque no suele ser necesario desarrollar código para procesar ficheros pcap, puesto que ya existe una infinidad de herramientas para trabajar con ellos. Por ejemplo, utilidades como 'mergecap' [mergecap] (que forma parte del paquete 'Wireshark'), que permiten mezclar el contenido de varios ficheros pcap en uno solo (e.g. si la captura de cada interfaz se almacena en un fichero diferente).

En la mayoría de los sistemas la captura de tráfico requiere permisos de superusuario, incluso cuando no se necesita poner el interfaz de red en modo promiscuo, por lo que 'tcpdump' se suele ejecutar como administrador o con 'sudo'. Aunque se puede utilizar la opción "-Z <user>" para que 'tcpdump' se convierta en el usuario indicado una vez que abre los interfaces de captura.

La opción para almacenar los paquetes capturados en un fichero pcap en lugar de mostrarlos por pantalla es '-w <fichero>'. Además de capturas de corta duración, como las que nos suelen interesar a nosotros para analizar posteriormente, 'tcpdump' también se puede usar para realizar capturas de larga duración (i.e. durante horas o incluso días), y existen más opciones como '-C', '-G', '-W' que hacen que 'tcpdump' vaya creando nuevos ficheros de captura cuando se llega a un número de paquetes determinado, se excede cierto tiempo, e incluso se pueden ir borrando los ficheros antiguos automáticamente para no llenar el sistema de ficheros. Los ficheros de captura se pueden leer con la opción '-r', lo que permite comprobar rápidamente su contenido, aunque, como ya hemos comentado, usaremos otras herramientas para la parte del análisis.

La opción para controlar de qué interfaz se desea capturar tráfico es '-i <interfaz>'. Para saber qué interfaces del sistema se pueden usar para capturar tráfico, y el tipo de interfaz que es, se puede utilizar el comando 'tcpdump -DL' (ver **Listado 3.14**).

**Listado 3.14** – Información de 'tcpdump' sobre los interfaces de captura

```
$ tcpdump -DL
1.eth0 [Up, Running]
2.lo [Up, Running, Loopback]
3.any (Pseudo-device that captures on all interfaces) [Up, Running]
4.bluetooth-monitor (Bluetooth Linux Monitor) [none]
5.nflog (Linux netfilter log (NFLOG) interface) [none]
6.nfqueue (Linux netfilter queue (NFQUEUE) interface) [none]
```

En las últimas versiones de 'tcpdump' y de los sistemas operativos, puede aparecer el interfaz virtual 'any' que permite capturar simultáneamente de todos los interfaces de red del equipo. Cuando se captura de un interfaz de red, por defecto se capturan tanto las tramas entrantes como las salientes. La dirección del tráfico capturado se puede controlar con la opción --direction={in|out|inout}. Además, por defecto 'tcpdump' intenta poner el interfaz de captura en modo promiscuo (excepto si se usa el interfaz '*any*'), si no se desea ese comportamiento se puede emplear la opción '-p'.

Otro parámetro fundamental de 'tcpdump' es la cantidad de bytes que se capturan de cada paquete. En varias versiones de 'tcpdump', por defecto solo se capturaban los primeros 64 octetos de cada trama, lo que permitía inspeccionar las cabeceras Ethernet, IP, TCP/UDP/ICMP y los primeros bytes de datos, pero evitaba capturar la mayoría de los datos a nivel de aplicación, que es donde pueden estar los datos más sensibles para los usuarios. Para analizar el comportamiento de una aplicación típicamente es necesario capturar todo el tráfico que intercambia, por lo que es recomendable usar la opción '-s0', que permite capturar todos los datos de los paquetes. Pero para preservar la privacidad de los usuarios de la red, además de para reducir la cantidad de tráfico capturado, se recomienda indicar explícitamente el tráfico que se desea capturar mediante un filtro.

Y es que 'tcpdump' posee un potente lenguaje de filtrado para elegir qué tráfico quiere capturarse. Existen decenas de primitivas [pcap-filter] sobre toda clase de protocolos (incluso se pueden aplicar reglas sobre los datos transportados), y además se pueden combinar con expresiones lógicas. Esas expresiones utilizan una sintaxis denominada *Berkeley Packet Filter* (BPF), y que en ciertos sistemas operativos se puede evaluar a nivel de núcleo (*kernel*) en lugar de en espacio de usuario, de forma que tiene un rendimiento excelente.

En nuestro caso nos ceñiremos en las primitivas para seleccionar todo el tráfico, pero únicamente de un sistema porque, siempre que haya suficiente capacidad,

normalmente es preferible capturar todo el tráfico y luego filtrarlo cuando se analiza. Para ello se puede utilizar la expresión "host <IP addr>" para capturar todos los datagramas enviados o recibidos por esa dirección IP (las expresiones "src host <addr>", y "dst host <addr>" permiten elegir únicamente los datagramas con las direcciones origen o destino indicadas), aunque si se conoce la dirección MAC del sistema sería preferible filtrar con "ether host <MAC addr>" (o sus variantes "ether src <addr>", "ether dst <addr>"), porque así también se capturaría tráfico no-IP como por ejemplo ARP. Otras primitivas que pueden ser útiles para centrarnos en protocolos y puertos específicos serían "arp", "icmp", "tcp", "udp" y "port <puerto>" (y sus variantes "src port <puerto>" y "dst port <puerto>"). Por ejemplo, la expresión "dst host www.example.com and tcp and dst port 80" solo capturaría el tráfico destinado al puerto 80 del servidor www.example.com.

En resumen, el uso más habitual que vamos a hacer de 'tcpdump' para capturar todo el tráfico del sistema que ejecuta la aplicación a analizar será: "sudo tcpdump -i any -s0 -p -w captura_local.pcap" cuando se capture en el propio equipo, y "sudo tcpdump -i <iface> -s0 -w captura_remota.pcap ether host <MAC addr>" cuando la captura se realice desde una estación en la misma subred.

### 3.2.2. Análisis de tráfico con 'Wireshark'

'Wireshark' [Wireshark], antes conocido como 'Ethereal', es una herramienta gráfica de captura y análisis de tráfico de red. Además, incluye varias herramientas adicionales, como 'tshark' [tshark] que es muy similar a 'tcpdump', puesto también permite capturar y visualizar tráfico desde la línea de comandos, pero tiene incluso más opciones de visualización que éste.

La principal característica de 'Wireshark' es que es capaz de interpretar casi cualquier protocolo conocido. Si alguien diseña un protocolo nuevo, una de las primeras cosas que hará será desarrollar un módulo de 'Wireshark' para poder analizarlo y depurarlo. Sin embargo, la calidad de muchos de estos *plugins* deja mucho que desear (sobre todo con paquetes mal formados), así que 'Wireshark' es una fuente inagotable de vulnerabilidades de *buffer overflow*. Por lo tanto, no es nada recomendable ejecutar 'Wireshark' con permisos de administrador, y por lo tanto capturar directamente con él, más allá que para realizar pruebas rápidas en entornos controlados.

Cuando se abre un fichero de captura en 'Wireshark' (Archivo > Abrir), la pantalla principal (**Figura 3.15**) está dividido en tres paneles. El panel superior muestra una lista de todos los paquetes capturados. Cuando se selecciona uno de esos paquetes, el panel del centro muestra el análisis de todas las cabeceras de protocolo que contiene el mismo, donde es posible consultar el valor (interpretado) de todos los

campos de cada una de ellas (e incluso metainformación, como la relación de ese paquete otros paquetes), mientras que el panel del abajo muestra el valor de todos los octetos del paquete, tanto en representación hexadecimal (izquierda) como en texto (derecha). Si se selecciona uno de los campos en el panel central, los octetos de dicho campo se resaltan en el panel inferior.

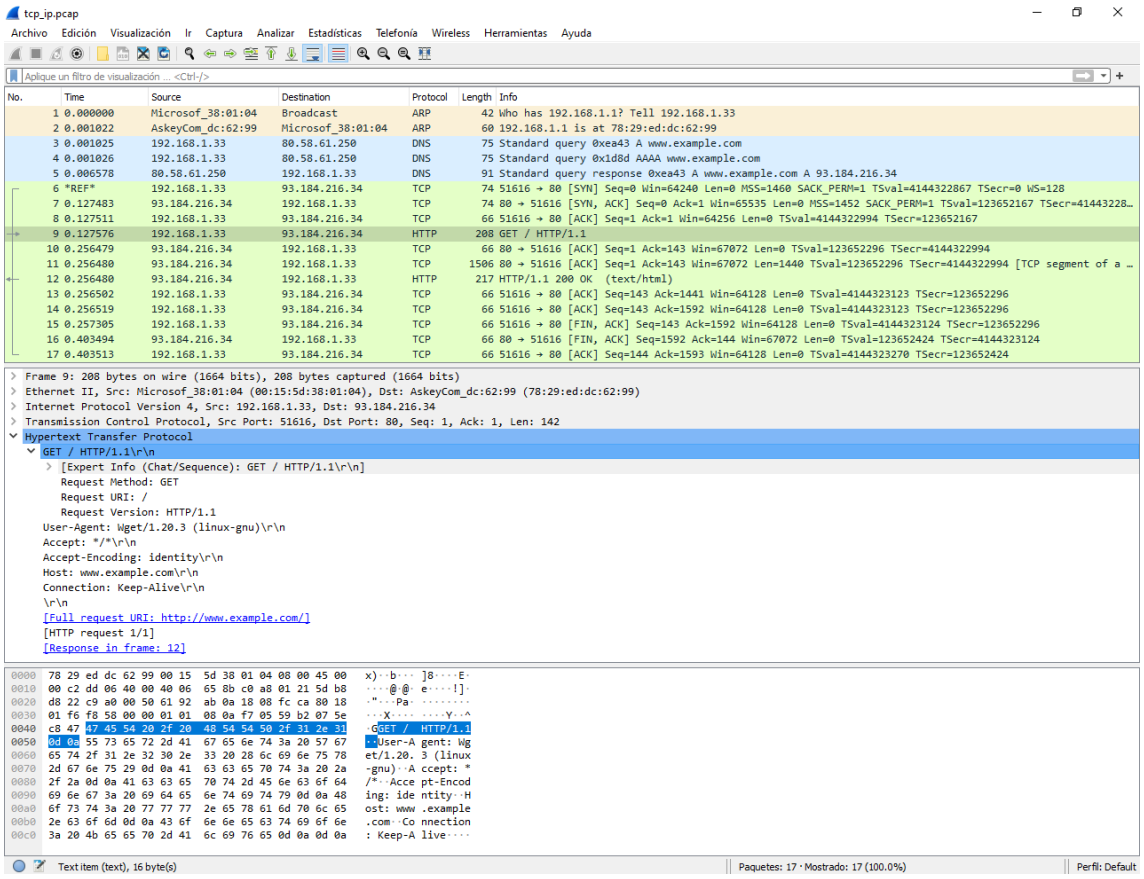



**Figura 3.15** – Pantalla principal de 'Wireshark'

Aunque el listado de paquetes capturados solo se muestra en la pantalla principal de 'Wireshark', haciendo doble *click* en cualquier paquete de la lista abre una ventada adicional (**Figura 3.16**) con la información de los paneles central e inferior del mismo. De esta forma se pueden visualizar dos o más paquetes simultáneamente, y así comparar una petición con su respuesta, qué campos varían en mensajes consecutivos, etc.

Las primeras columnas de la lista de paquetes son el número de paquete y el *timestamp* de cuando se capturó, y que por defecto muestra el número de segundos que han pasado desde el principio de la captura. Aunque esta información permite hacerse una idea general de cuando ocurren los diferentes eventos de red dentro de la captura, para calcular con más exactitud la temporización relativa de los paquetes (e.g. para calcular el RTT) puede ser útil seleccionar un paquete y establecer una referencia de tiempo en él (en el menú contextual que aparece al

hacer *click* con el botón derecho del ratón o dentro del menú "Edición"), de forma que todos los paquetes a partir de ese muestran como *timestamp* el tiempo relativo desde la última referencia de tiempo. En el mismo menú contextual de un paquete se pueden marcar paquetes importantes, ignorar los que no lo son, añadir un comentario, configurar la resolución de nombres, colorear conversaciones, y generar filtros de visualización.

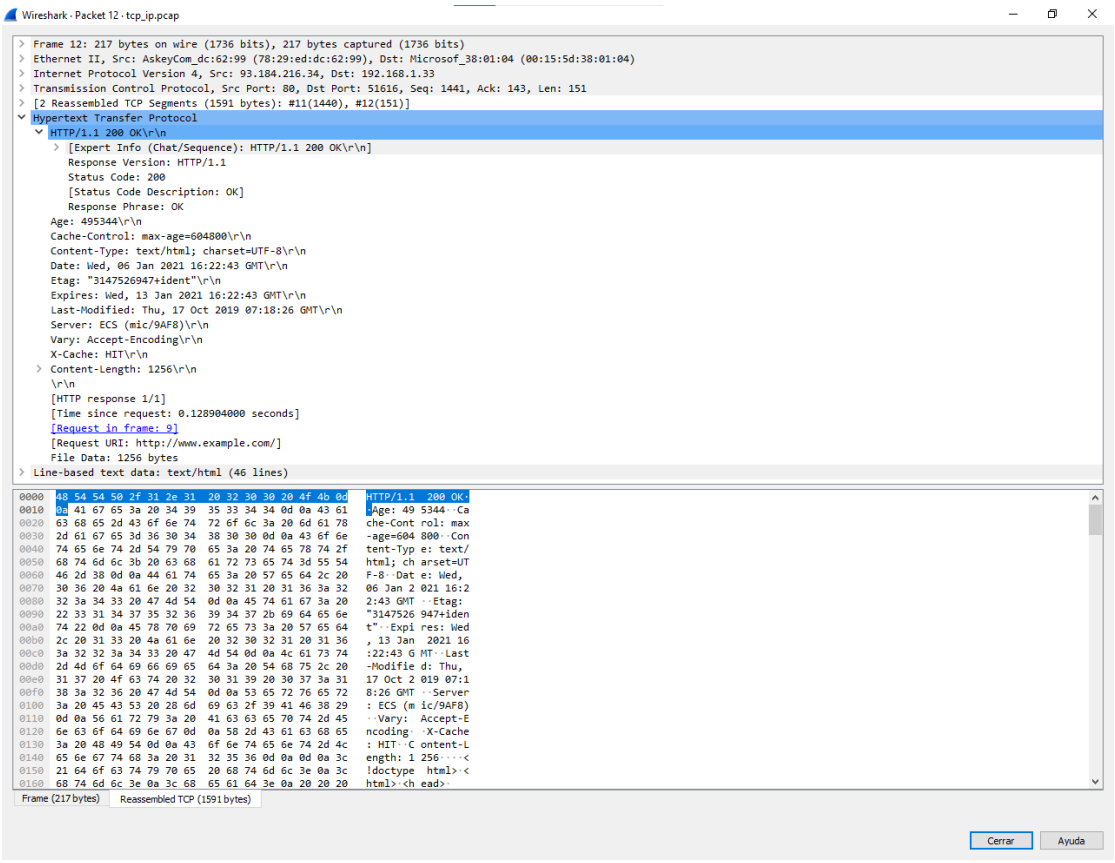



**Figura 3.16** – Pantalla adicional con información sobre un paquete

Como normalmente en los ficheros de captura de tráfico hay una gran cantidad de paquetes, 'Wireshark' dispone de un potente motor de filtrado que permite visualizar en la lista de paquetes únicamente aquellos que cumplen cierta expresión de búsqueda. Las expresiones de los filtros de búsqueda se pueden introducir directamente en la barra de texto que hay justamente encima de la lista de paquetes. La sintaxis es diferente a los filtros de captura BFP de 'tcpdump', pero son incluso más potentes, ya que permiten buscar prácticamente por cualquier campo de cualquier protocolo (**Figura 3.17**). La expresiones de los filtros de visualización de 'Wireshark' también admiten operadores lógicos ('==', 'not', '!', 'and', '&&', 'or', '||') y normalmente están formados por operandos del tipo '<*protocolo*>.<*campo*>' , como por ejemplo 'ip.src', 'ip.dst' o 'ip.addr' para referirse a la dirección IP origen, destino, o cualquiera de ellas.

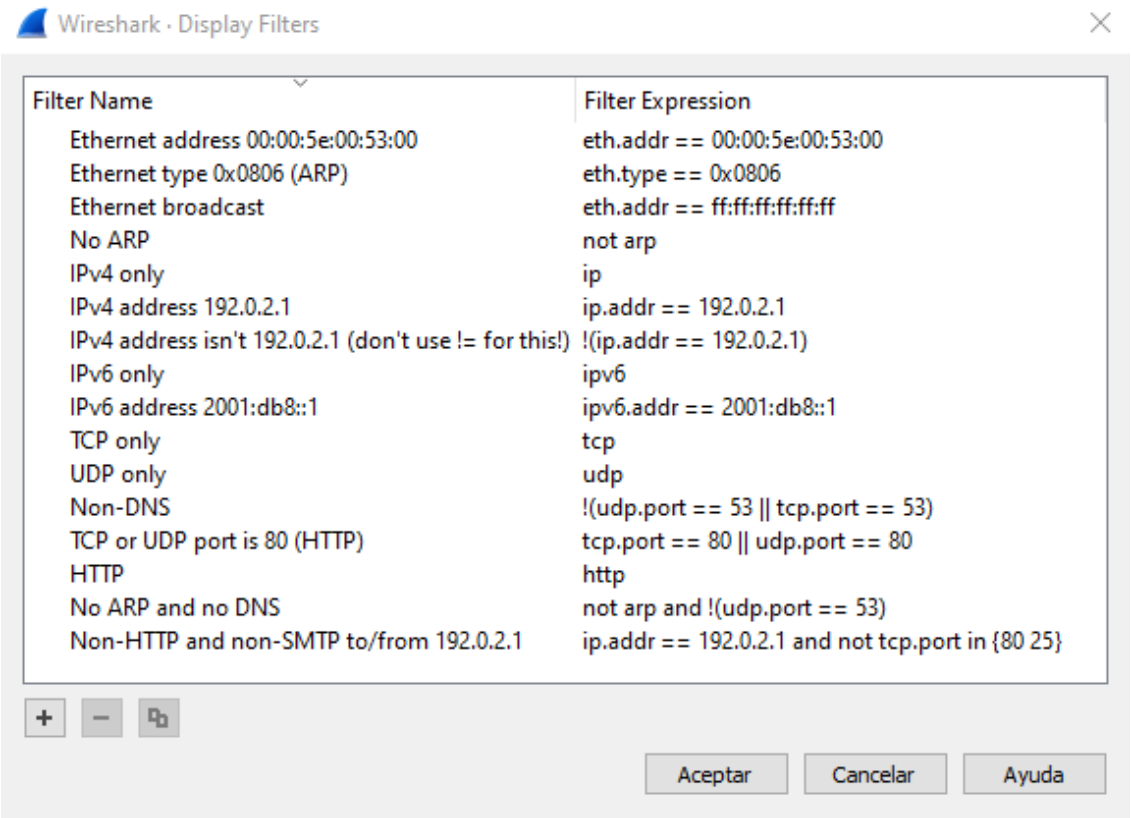



**Figura 3.17** – Ejemplos de filtros de visualización de 'Wireshark'

Sin embargo, no es necesario aprenderse la multitud de opciones que tiene, puesto que el campo de la barra de filtros de visualización tiene funciones de autocompletado. Al escribir el nombre de un protocolo seguido de '.' se despliega la lista de las expresiones disponibles para ese protocolo (así como las últimos filtros utilizados), y según se siguen tecleando letras, se van reduciendo las opciones disponibles (pruebe a teclear 'tcp.' y luego 'tcp.flags.'). Además, para simplificar el diseño de filtros complejos, la barra cambia de color de verde a rojo, en función de si la expresión es correcta o incorrecta.

Nótese la diferencia entre los filtros de visualización, que se usan únicamente para seleccionar qué paquetes se muestran en la lista de paquetes (y que usan esta sintaxis específica de 'Wireshark'), y los filtros de captura, que se pueden especificar en la pantalla de captura de 'Wireshark' (menú "Captura"), utilizan la misma sintaxis BFP que 'tcpdump', y que como en 'tcpdump' limitan los paquetes que se capturan, así que si un error en el filtro de captura puede requerir repetir la captura porque no se dispone de la información necesaria, mientras que un error en un filtro de visualización no hace perder información, y solo basta con escribirlo de nuevo para encontrar la información deseada. Así que, a no ser que ya se tenga muy caracterizada una aplicación y se sepa exactamente qué información capturar, la recomendación es que solo se usen los filtros de captura (tanto en 'Wireshark' como en 'tcpdump') para capturar todo el tráfico del sistema que ejecuta la aplicación bajo

análisis (utilizando los filtros 'host' y 'ether host') y filtrar el resto de estaciones de la misma subred para reducir la cantidad de paquetes de la captura a analizar y salvaguardar la privacidad del resto de usuarios de la subred. Una vez que todo el tráfico de la estación está en el fichero de captura, se pueden usar los filtros de visualización de 'Wireshark' para buscar la aplicación.

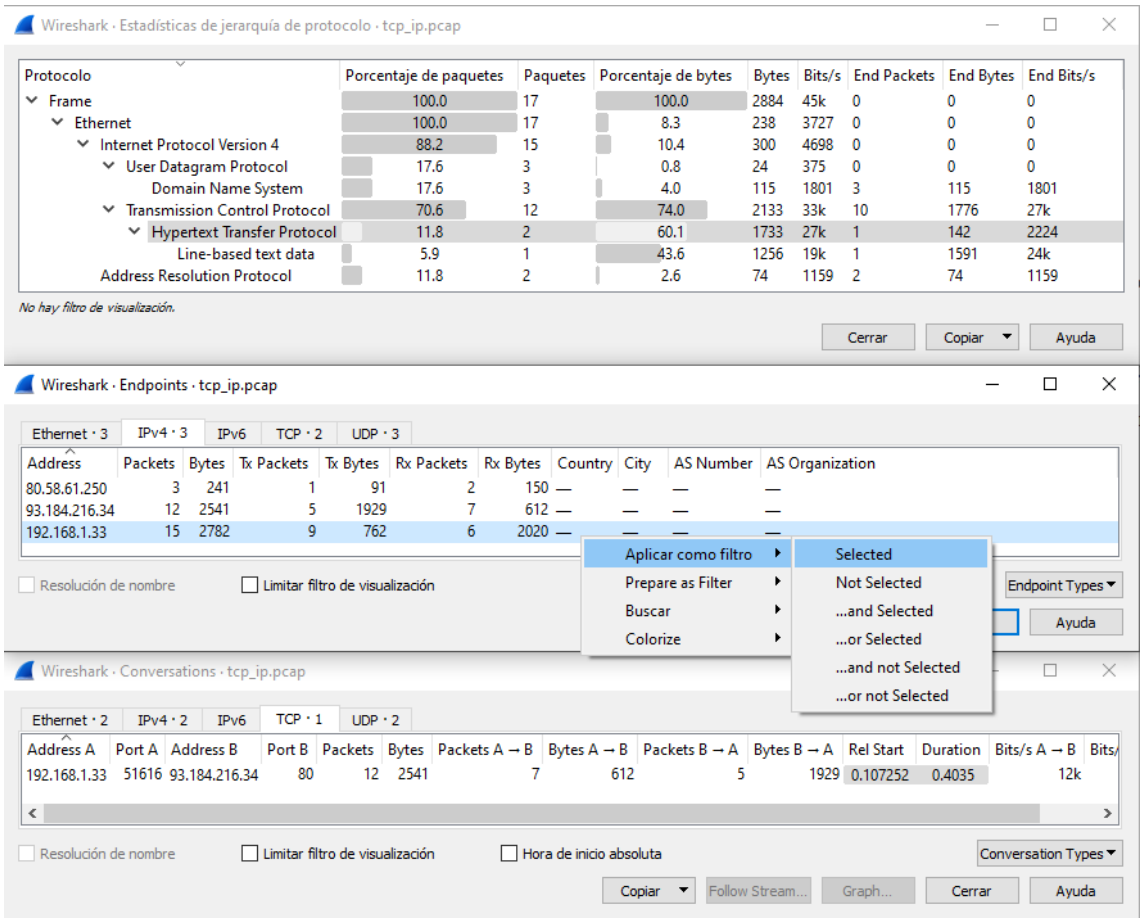



**Figura 3.18** – Diferentes estadísticas de 'Wireshark'

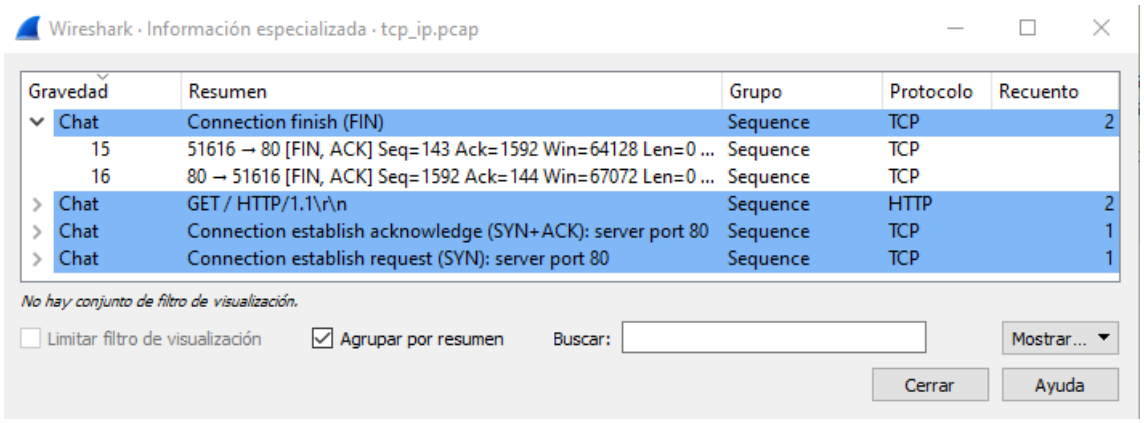



**Figura 3.19** – Información especializada de un fichero de captura

En este sentido, el menú "Estadísticas" proporciona resúmenes muy interesantes sobre el fichero de captura global, tales como los protocolos, sistemas finales y conversaciones detectadas (**Figura 3.18**). Además, permite crear filtros de visualización para cualquiera de los elementos que aparece en ellos, simplemente pulsando con el botón derecho del ratón sobre ellos (se puede hacer lo mismo sobre cualquier campo o paquete que aparezca en las pantallas de 'Wireshark'). Además, el menú "Analizar > Información especializada" (**Figura 3.19**) permite identificar rápidamente los principales eventos de red o aquellos más inusuales dentro de todo el fichero de captura.

Una vez que se ha identificado el tráfico de la aplicación, hay un par de pantallas que simplifican mucho en análisis de conexiones TCP completas, en lugar de ir analizando paquete a paquete. La primera está en "Analizar > Seguir > Flujo TCP" y, cuando se tiene seleccionado un paquete TCP (también se puede acceder desde el menú contextual del paquete TCP), crea automáticamente un filtro de visualización que sólo muestra los paquetes de esa conexión TCP, y además muestra en otra pantalla (**Figura 3.20**) todos los datos intercambiados dentro de esa conexión, marcando en rojo los datos enviados por el cliente (i.e. quién inicia la conexión TCP) y en azul los datos que devuelve el servidor, y que además se pueden almacenar en un fichero.

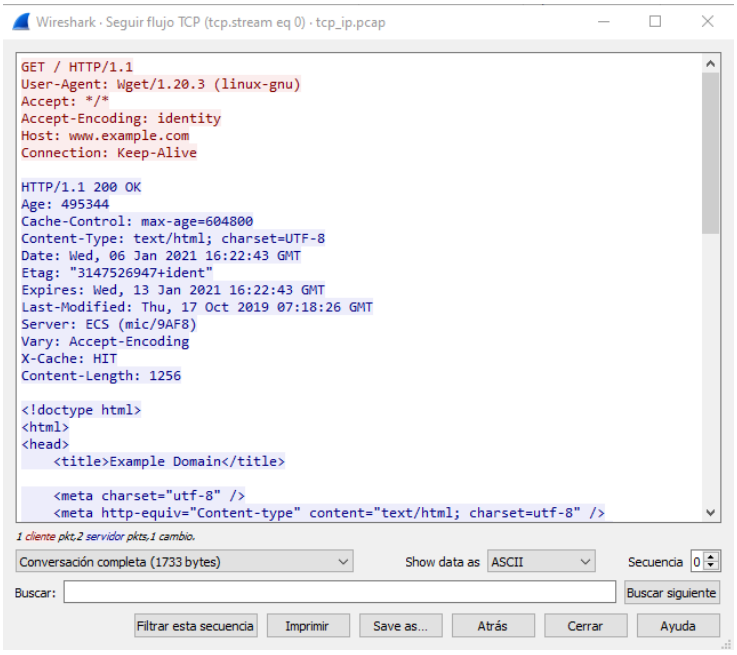



**Figura 3.20** – Contenido intercambiado en una conexión TCP

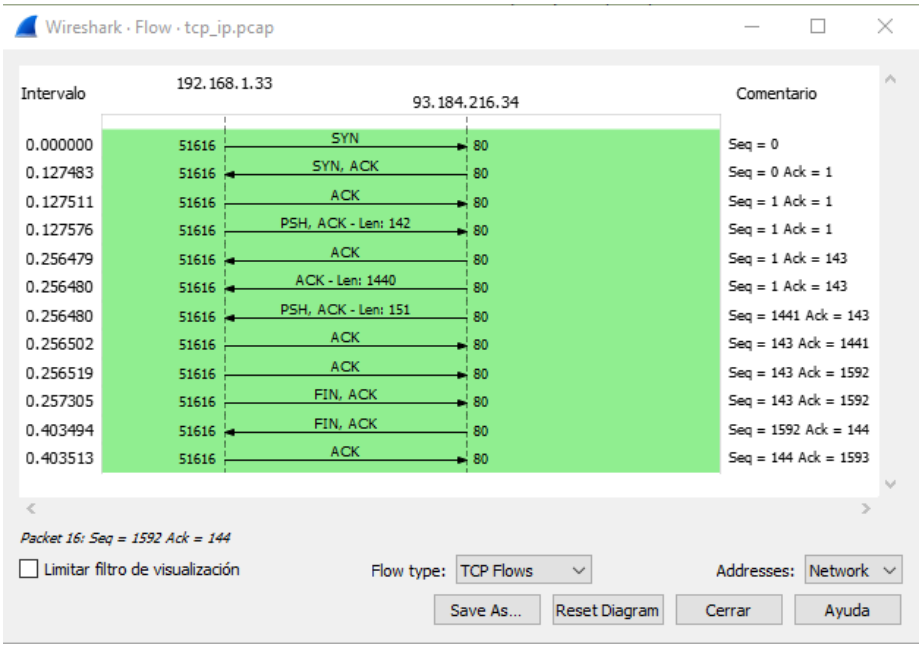



**Figura 3.21** – Gráfica de flujo de una conexión TCP

Si además del contenido se quiere analizar la conexión TCP en sí, la opción "Estadísticas > Gráfica de flujo" muestra un diagrama de flujo (**Figura 3.21**) con la información más importante de los paquetes que forman una conexión TCP o UDP.

En el menú "Visualización", además de controlar todos los detalles sobre el aspecto gráfico de 'Wireshark' y de los paquetes mostrados, hay una opción para controlar la resolución de nombres, que merece cierto comentario. 'Wireshark' es capaz de asignar resolver los nombres asociados a los puertos (e.g. 80 → http), las direcciones MAC (resolviendo el OUI del fabricante), y resolver el nombre DNS asociado a las direcciones IP. Hay que tener cuidado con esta última opción, puesto que, aunque puede ser muy útil para identificar los extremos de las comunicaciones, también puede generar tráfico adicional no deseado (especialmente si se utiliza el propio 'Wireshark' para realizar la captura de tráfico). Además de permitir al usuario especificar un nombre para cualquier dirección IP (en "Menú contextual > Editar nombre resuelto"), 'Wireshark' es capaz de analizar el tráfico DNS dentro del fichero de captura, y asignar a las IPs resueltas el nombre por el que se preguntó. Estos dos métodos de resolución son pasivos y pueden ser muy útiles. Sin embargo, si se activa la opción "Edición > Preferencias... > *Name Resolution* > *Use an external network name resolver*" (**Figura 3.22**), 'Wireshark' realizará una consulta DNS inversa para cada dirección IP no pueda resolver con los métodos anteriores. Además de generar una gran cantidad de tráfico DNS (que podría llegar a los servidores DNS de las organizaciones destino de las comunicaciones, lo que las podría poner en sobre aviso de que se están analizando sus comunicaciones), las resoluciones DNS inversas normalmente no devuelven nombres demasiado significativos, así que se recomienda desactivar esta opción.

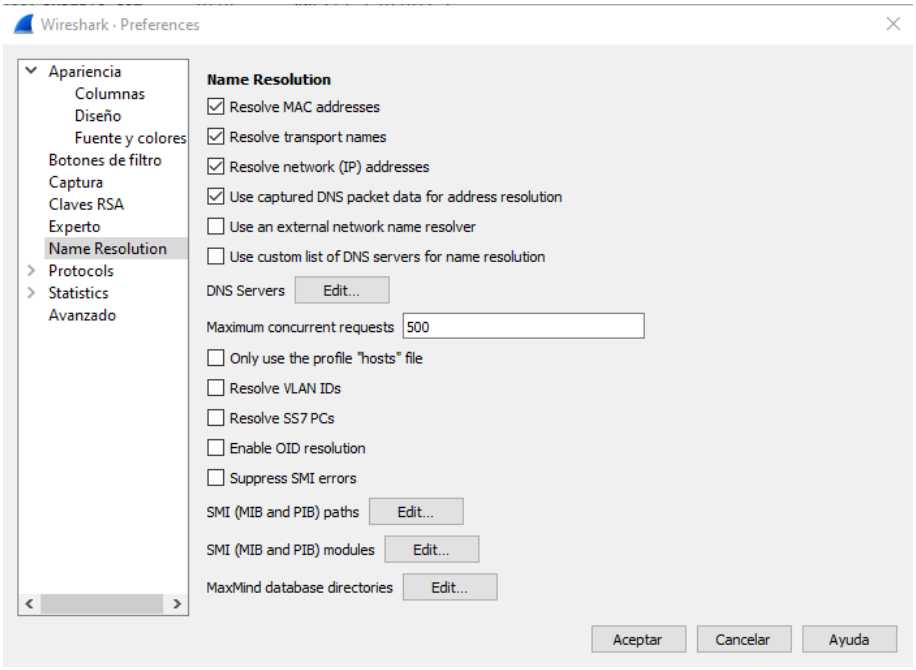



**Figura 3.22** – Preferencias del protocolo TCP en 'Wireshark'

Esto es solo un pequeño resumen de las opciones más comunes y útiles de 'Wireshark' para el análisis de protocolos, pero tiene muchísimas más, y que permiten analizar en profundidad protocolos de lo más variopintos (ver menús "Telefonía" y "Wireless"). Para hacerse una idea de la cantidad de protocolos soportados, y las opciones de visualización y análisis de cada uno de ellos, merece la pena echar un vistazo sobre "Edición > Preferencias … > Protocolos" (**Figura 3.23**), y en general recorrer los diferentes menús de 'Wireshark' para ver todas las capacidades de análisis que proporciona.

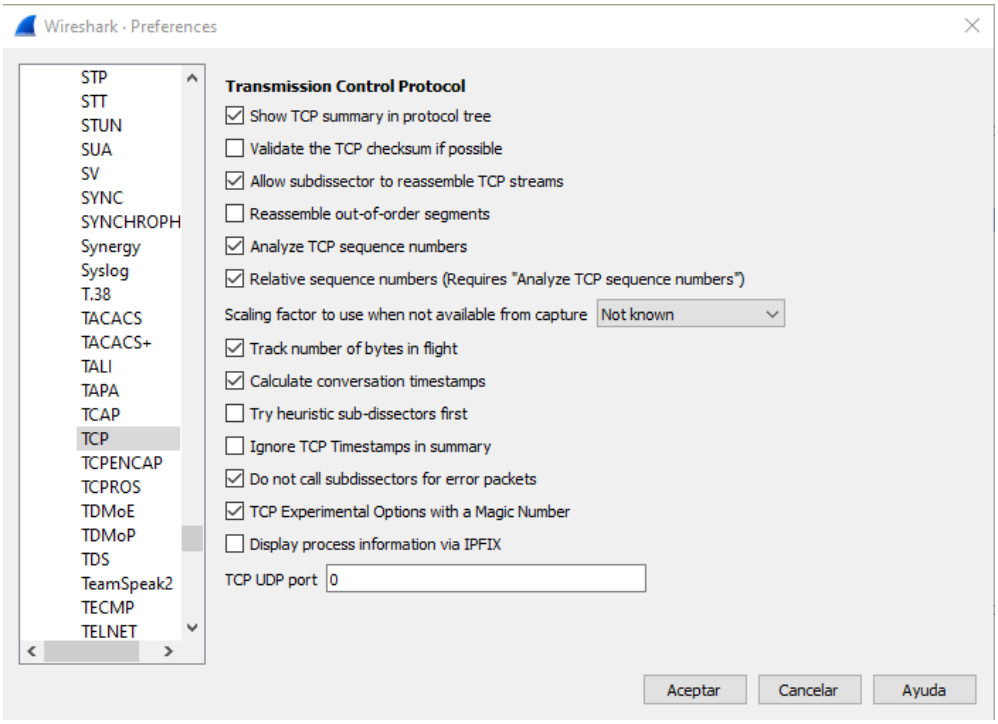



**Figura 3.23** – Preferencias del protocolo TCP en 'Wireshark'

### 3.2.3. Análisis de tráfico a nivel de aplicación con 'mitmproxy'

Aunque 'tcpdump' y 'Wireshark' son las herramientas de referencia para la captura y análisis de tráfico a nivel de red, en algunos escenarios (e.g. si no estamos en la misma subred que el sistema a analizar, no podemos observar su tráfico, o se emplea TLS) es necesario desplegar un *proxy* a nivel de aplicación. Existen un gran número de *proxies* que permiten analizar el tráfico a nivel de aplicación (especialmente HTTP), como por ejemplo 'Burp Suite' [BurpSuite], que además incluye un gran número de herramientas y opciones para hacer *pentesting* de aplicaciones web. Sin embargo, debido a su complejidad y al hecho de que de manera nativa solo soporte tráfico HTTP, vamos a utilizar en su lugar una herramienta mucho más simple: 'mitmproxy'.

mitmproxy [mitmproxy] es una herramienta muy flexible, que permite operar como un *proxy* HTTP convencional, como un *proxy* HTTP transparente, como un *proxy* HTTP inverso, o como un *proxy* SOCKS5. Además, permite la interceptación de tráfico cifrado con TLS, aunque describiremos la capacidad de capturar tráfico TLS con 'mitmproxy' en el **Apartado 7.3**.

Aunque 'mitmproxy' tiene un gran número de parámetros y opciones, las más importantes son el modo de funcionamiento (--mode {"regular"|"transparent"|"socks5"|"reverse: *<Server URL>*"}), y la dirección (--listen-host *<IP addr>*)

y puerto (--listen-port <*port*>) desde la que escuchará las peticiones de los clientes (**Listado 3.24**).

**Listado 3.24** – Ejecución de la herramienta 'mitmproxy' en modo *proxy* HTTP.

```
$ mitmproxy --mode regular --listen-host 127.0.0.1 --listen-port 8080
```

Si ahora lo configuramos en el navegador como *proxy* cliente, y visitamos una web HTTP (e.g. http://ww.example.com), podemos ver en la consola de 'mitmproxy' un listado de todos los mensajes HTTP (**Figura 3.25**).


**Figura 3.25** – Lista de peticiones y respuestas capturadas por 'mitmproxy'

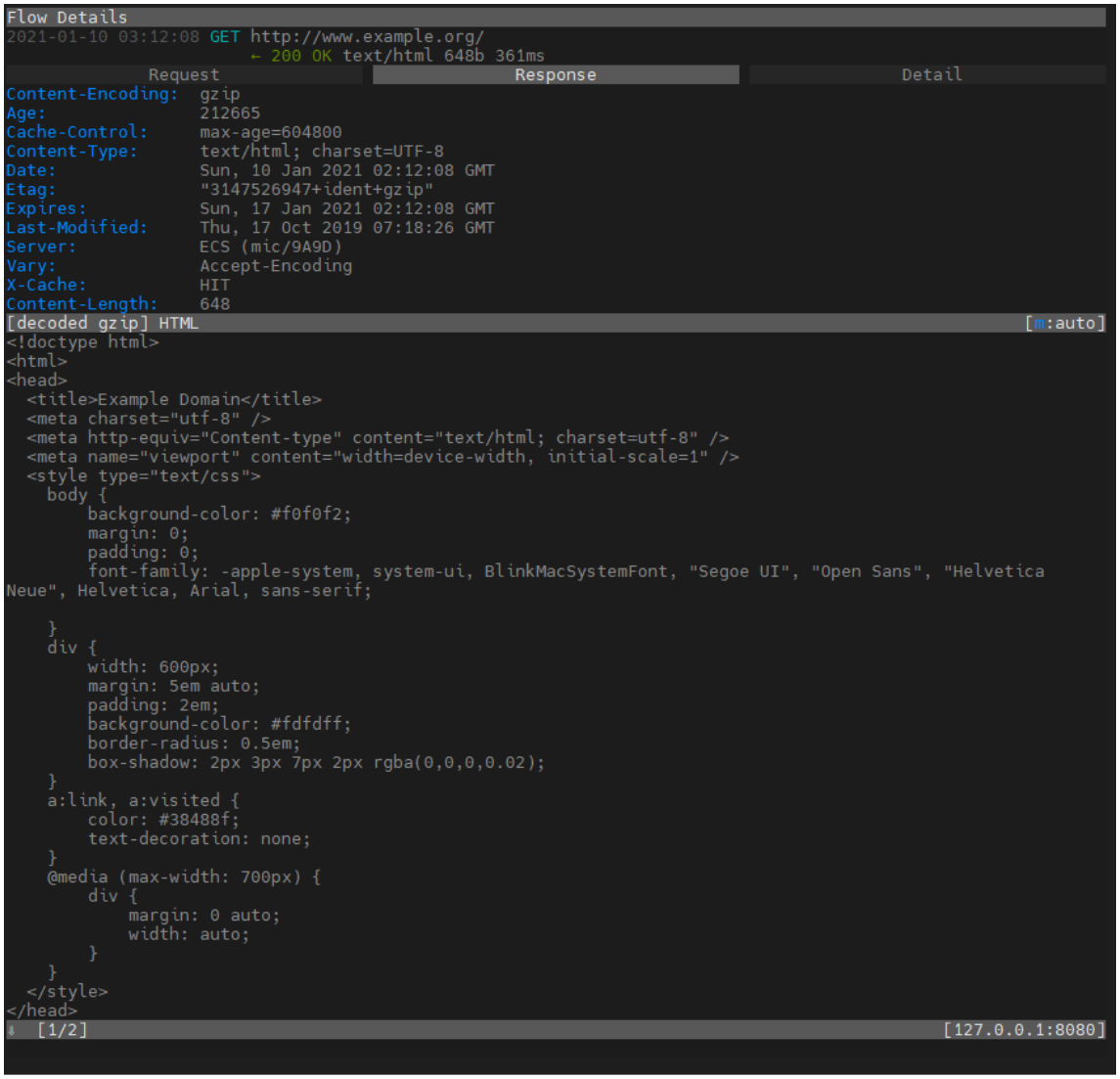



**Figura 3.26** – Detalle de una petición-respuesta HTTP capturada por 'mitmproxy'

Aunque el interfaz de 'mitmproxy' parece muy básico, es muy funcional. Se pueden ver todos los detalles sobre un petición-respuesta HTTP (*flows* en su terminología) simplemente eligiéndola con los cursores y pulsando [Enter], o simplemente haciendo *click* con el ratón sobre ella. Se pueden ver las cabeceras y cuerpo del mensaje de la petición, la respuesta (**Figura 3.26**), y datos adicionales sobre el servidor y el cliente. Para volver a la pantalla principal con la lista de flujos HTTP, basta con pulsar 'q' (que también sirve para salir del programa). Al pulsar la tecla '?' muestra todos los comandos del interfaz, incluyendo las expresiones de búsqueda y los comandos para cargar y salvar la información sobre los flujos a fichero.

Otros interfaces alternativos de 'mitmproxy' son: 'mitmweb' que tiene un interfaz web para ver los mensajes capturados, y 'mitmdump' que muestra los mensajes HTTP capturados directamente en la línea de comandos (como 'tcpdump').

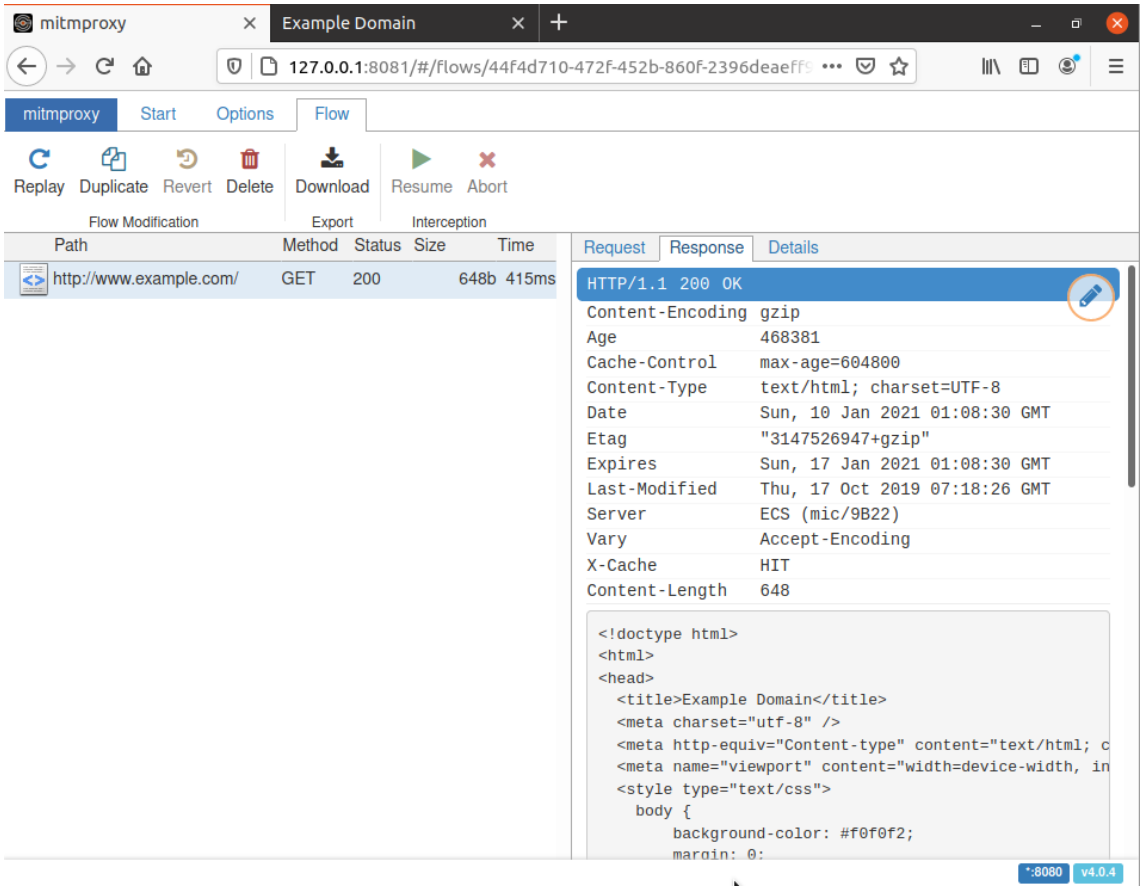



**Figura 3.27** – Interfaz web de 'mitmweb'

Para trabajar en modo *proxy* inverso (**Listado 3.28**) basta con indicar a qué servidor se enviarán todas las peticiones que reciba. Como habrá que escuchar en el puerto 80 (i.e. http), que es un puerto bien conocido, es necesario lanzar el *proxy* con permisos de superusuario.

**Listado 3.28** – Ejecución de la herramienta 'mitmproxy' en modo *proxy* inverso.

```
$ sudo mitmproxy --mode reverse:http://www.example.com --listen-host 0.0.0.0
--listen-port 80
```

# 4. Análisis de aplicaciones basadas en el protocolo UDP

En este capítulo estudiaremos cómo analizar aplicaciones que utilizan el protocolo UDP (*User Datagram Protocol*). Así que se recomienda repasar el apartado sobre el protocolo UDP en el **Capítulo 2** de Repaso de la torre de protocolos TCP/IP.

## 4.1. Análisis de una aplicación cliente UDP

En este apartado amos a analizar una aplicación cliente-servidor UDP de ejemplo. Para ello empezaremos asumiendo que tenemos acceso al código fuente de la aplicación. Luego, solo al ejecutable de la misma, y finalmente solo al tráfico de red que genera la misma. El objetivo es ir asociando los diferentes eventos que ocurren en las diferentes trazas con el comportamiento de la aplicación. El código fuente completo de todos los ejemplos se distribuye junto a esta memoria, por lo que se recomienda compilarlo y ejecutarlo para analizar el resultado de las diferentes herramientas.

La aplicación cliente-servidor UDP que analizaremos es muy simple. El cliente envía un mensaje de texto al servidor UDP y espera una respuesta de éste para imprimirla. Como UDP es un protocolo no fiable, si el cliente no recibe respuesta de servidor puede enviar hasta 3 veces el mensaje al servidor hasta que reciba respuesta. El servidor UDP simplemente recibe mensajes de un cliente UDP y se lo reenvía tal cual (como haría un servidor de Echo - UDP/7).

Así que empecemos analizando el programa cliente. El **Listado 4.1** muestra el código fuente en C del programa, mientras que los **Listados 4.2, 4.3 y 4.4** muestran sendas ejecuciones de dicho programa.

**Listado 4.1** – Código fuente editado del cliente UDP de ejemplo (udp_client.c)

```
#define BUFFER_SIZE 1024
#define TIMEOUT 3000 /* ms = 3 sec */
#define MAX_RETRIES 2

int main ( int argc, char* argv[] )
{
  if (argc != 4) {
    const char * myself = basename(argv[0]);
    fprintf(stderr, "Usage: %s <addr> <port> <msg>\n", myself);
    fprintf(stderr, " <server>: Domain name or IP address of the UDP
                      server\n");
    fprintf(stderr, "   <port>: Port number of the UDP server\n");
    fprintf(stderr, "    <msg>: Message to send to the UDP server\n");
    exit(EXIT_FAILURE);
```

```
}
const char* server_arg = argv[1];
const char*   port_arg = argv[2];
const char*    msg_arg = argv[3];

/* Obtain server address(es) matching host/port */
struct addrinfo hints;
memset(&hints, 0x00, sizeof(hints));
hints.ai_family = AF_UNSPEC;     /* Allow IPv4 or IPv6 addresses */
hints.ai_socktype = SOCK_DGRAM;  /* Datagram socket */
hints.ai_flags = AI_ADDRCONFIG;
hints.ai_protocol = IPPROTO_UDP; /* UDP protocol */
struct addrinfo *result;
int err = getaddrinfo(server_arg, port_arg, &hints, &result);
if (err != 0) {
  fprintf(stderr, "getaddrinfo() error: %s\n", gai_strerror(err));
  exit(EXIT_FAILURE);
}

/* getaddrinfo() returns a list of address structures. Try each
 * address until we successfully connect(). If socket() of connect()
 * fails, close the socket and try the next address. */
struct addrinfo *ptr;
int sock_fd;
for (ptr = result; ptr != NULL; ptr = ptr->ai_next) {
  /* Open an UDP socket */
  sock_fd = socket(ptr->ai_family, ptr->ai_socktype,
                   ptr->ai_protocol);
  if (sock_fd == -1) {
    continue; /* Failure! Try next address */
  }
  /* Fix socket to server address and UDP port */
  err = connect(sock_fd, ptr->ai_addr, ptr->ai_addrlen);
  if (err != -1); {
    break; /* Success! No need to keep looking for more addresses */
  }
  close(sock_fd); /* Close socket, because connect() failed */
}

if (ptr == NULL) {
  /* No address succeeded */
  fprintf(stderr, "Cannot connect to UDP Server '%s/%s'\n",
          server_arg, port_arg);
  exit(EXIT_FAILURE);
}

freeaddrinfo(result); /* No longer needed */

/* Get Server IP address and port */
```

```
struct sockaddr_storage server_sa;
socklen_t server_sa_len = sizeof(server_sa);
err = getpeername(sock_fd, (struct sockaddr *)
            &server_sa, &server_sa_len);
if (err == -1) {
  perror("getpeername() error");
  exit(EXIT_FAILURE);
}

char server_addr_str[NI_MAXHOST];
char server_port_str[NI_MAXSERV];
int gni_flags = NI_NUMERICHOST | NI_NUMERICSERV;
err = getnameinfo((struct sockaddr *) &server_sa, server_sa_len,
                  server_addr_str, sizeof(server_addr_str),
                  server_port_str, sizeof(server_port_str),
            gni_flags);
if (err == -1) {
   fprintf(stderr, "getnameinfo() error: %s\n", gai_strerror(err));
   exit(EXIT_FAILURE);
}

/* Send a message to the server until we get a reply */
char* msg = (char*) msg_arg;
int msg_len = strlen(msg);

nfds_t num_poll_fds = 1;
struct pollfd poll_fds[num_poll_fds];
poll_fds[0].fd = sock_fd;
poll_fds[0].events = POLLIN | POLLERR;

int retries;
for (retries=0; retries<=MAX_RETRIES; retries++) {
  ssize_t bytes_sent = send(sock_fd, msg, msg_len, 0);
  if (bytes_sent == -1) {
    perror("send() error");
    exit(EXIT_FAILURE);
  }
  printf("%ld bytes sent to UDP Server %s/%s\n",
              bytes_sent, server_addr_str, server_port_str);

  /* Now wait until we have a response.
   * We use poll() to implement the retransmission timer */
  int retval = poll(poll_fds, num_poll_fds, TIMEOUT);
  if (retval == -1) {
    perror("poll() error");
    exit(EXIT_FAILURE);
  } else if (retval == 0) {
    if (retries+1 <= MAX_RETRIES) {
       printf("Timeout! Retrying #%d\n", retries+1);
```

```
    }
    continue;
  } else {
    /* We have received a response from the server */
    break;
  }
}

if (retries > MAX_RETRIES) {
  /* No response from any of the replies. Abort! */
  fprintf(stderr, "Cannot contact UDP Server %s/%s\n",
                server_addr_str, server_port_str);
  exit(EXIT_FAILURE);
}

/* Receive the message from server */
char buffer[BUFFER_SIZE];
ssize_t bytes_recv = recv(sock_fd, buffer, sizeof(buffer) - 1, 0);
if (bytes_recv == -1) {
  perror("recv() error");
  exit(EXIT_FAILURE);
}

buffer[bytes_recv] = '\0';
printf("%ld bytes received from UDP Server: \"%s\"\n",
      bytes_recv, buffer);

/* Close socket and exit */
err = close(sock_fd);
if (err != 0) {
  perror("close() error");
  exit(EXIT_FAILURE);
}

return EXIT_SUCCESS;
}
```

**Listado 4.2** – Información de uso del cliente UDP de ejemplo (udp_client)

```
$ ./udp_client
Usage: udp_client <addr> <port> <msg>
 <server>: Domain name or IP address of the UDP server
  <port>: Port number of the UDP server
   <msg>: Message to send to the UDP server
```

**Listado 4.3** – Ejecución con éxito del cliente UDP de ejemplo (udp_client)

```
$ ./udp_client 127.0.0.1 7000 "Hola 127.0.0.1!"
15 bytes sent to UDP Server 127.0.0.1/7000
15 bytes received from UDP Server: "Hola 127.0.0.1!"
```

**Listado 4.4** – Ejecución fallida del cliente UDP de ejemplo (udp_client)

```
$ ./udp_client www.example.com 7000 "Hola www.example.com!"
21 bytes sent to UDP Server 93.184.216.34/7000
Timeout! Retrying #1
21 bytes sent to UDP Server 93.184.216.34/7000
Timeout! Retrying #2
21 bytes sent to UDP Server 93.184.216.34/7000
Cannot contact UDP Server 93.184.216.34/7000
```

### 4.1.1. Principales llamadas al sistema de un cliente UDP

Aunque el programa en sí es muy sencillo, contiene todos los elementos que deberían utilizar el código cliente de una aplicación UDP. En particular emplea las siguientes llamadas al sistema y funciones de red POSIX:

- **int getaddrinfo(const char *node, const char *service, const struct addrinfo *hints, struct addrinfo **res)** - Esta función permite obtener todas las direcciones de transporte (i.e. dirección IP y número de puerto) de un nodo (e.g. "www.example.com" o "localhost" o "127.0.0.1" o "::1") y servicio (e.g. "80" o "http"). Combina la funcionalidad de gethostbyname() y getservbyname() en un único interfaz, y permite trabajar tanto con direcciones IPv4 como con direcciones IPv6, así como con nombres de dominio DNS. Los resultados se devuelven en memoria dinámica, que hay que liberar con **freeaddrinfo()**.

- **int socket(int domain, int type, int protocol)** - Los *sockets* son una abstracción definida en el estándar POSIX para representar las comunicaciones de los procesos (por ejemplo utilizando protocolos de red), aunque en C se implementan como descriptores de ficheros, por lo que también se pueden utilizar funciones de manipulación de ficheros sobre ellos, tales como read(), write() o close(). Esta llamada permite crear un *socket* de un determinado dominio, tipo y protocolo. El dominio o familia de direcciones **AF_INET** se refiere a protocolos IPv4, AF_INET6 se refiere a protocolos IPv6, y la familia **AF_UNSPEC** empleada en la llamada getaddrinfo() permite trabajar con cualquiera de ellos en equipos *dual-stack* IPv4/IPv6. Los protocolos fiables como TCP (**IPPROTO_TCP**) utilizan sockets de tipo **SOCK_STREAM**, mientras que los protocolos como UDP (**IPPROTO_UDP**) que están basados en datagramas utilizan sockets de tipo **SOCK_DGRAM**. Al ser una aplicación cliente que no

requiere un número de puerto específico, no es necesario realizar un bind(), por lo que se empleará un puerto efímero (i.e. en el rango 49.152-65.535).

- **int connect(int sockfd, const struct sockaddr *addr, socklen_t addrlen)** – Normalmente en los protocolos orientados a conexión, esta función permite establecer una conexión con el destino (veremos un ejemplo en el apartado de TCP). En el caso de UDP esta llamada no genera ningún tipo de tráfico de red, sino que simplemente le indica al núcleo del sistema que este *socket* solo se empleará para comunicarse con la dirección de transporte indicada. De este modo se pueden emplear las funciones send() y recv() para enviar y recibir datos, en lugar de las funciones sendto() y recvfrom(), que requieren indicar la dirección de transporte destino/origen.

- **int getpeername(int sockfd, struct sockaddr *addr, socklen_t *addrlen)** – Esta función permite obtener la dirección de transporte del servicio remoto al que está conectado un *socket* TCP (o un socket UDP después de llamar a connect()).

- **int getnameinfo(const struct sockaddr *addr, socklen_t addrlen, char *host, socklen_t hostlen, char *serv, socklen_t servlen, int flags)** – Esta es la función inversa a getaddrinfo(), puesto que permite obtener dada una dirección de transporte binaria (struct sockaddr), la representación textual de su dirección IP (IPv4 o IPv6) y su número de puerto, e incluso realizar una consulta inversa al DNS para obtener su nombre de dominio, aunque en este caso se ha deshabilitado con el *flag* NI_NUMERICHOST. Combina por tanto la funcionalidad de las funciones gethostbyaddr() y getservbyport(), y permite trabajar indistintamente con direcciones IPv4 e IPv6.

- **ssize_t send(int sockfd, const void *buf, size_t len, int flags)** - Esta función permite enviar datos a través de un *socket* que ya haya establecido una conexión con un destino (ver connect()), por lo que no es necesario indicar su dirección de transporte destino (a diferencia de sendto()).

- **int poll(struct pollfd *fds, nfds_t nfds, int timeout) –** Uno de los principales problemas del protocolo UDP es que no es fiable. Esto quiere decir que el mensaje que envía el cliente al servidor puede perderse (o la respuesta), y UDP no proporciona ningún mecanismo para detectar el fallo o recuperarlo. Por lo tanto, si simplemente nos quedásemos esperando a recibir con recv() y la petición o la respuesta se pierden, el programa se podría quedar esperando indefinidamente. Por esta razón, antes de recibir se utiliza la función poll(), que normalmente se utiliza para ver si un conjunto de descriptores de fichero tiene datos listos para ser leídos (i.e. como select(), aunque poll() es más escalable), aunque en este caso se emplea simplemente

para implementar un temporizador. De forma que, si no recibe respuesta del servidor en 3 segundos, se vuelve a enviar el mensaje (hasta 2 veces más).

- **ssize_t recv(int sockfd, void *buf, size_t len, int flags)** – Una vez que poll() nos indica que hay datos disponibles en el *socket* UDP, podemos obtenerlos mediante la llamada al sistema recv(). UDP es un protocolo orientado a datagrama, por lo que cuando se reciben datos, estos forman un mensaje completo, por lo que no suele ser necesario definir mecanismos adicionales para marcar el inicio y fin de los mensajes (como sí ocurre en TCP).

- **int close(int fd)** – Una vez que el cliente ha recibido respuesta del servidor UDP, puede cerrar el socket. Para ello utiliza la misma función close() que se emplearía para cerrar un fichero abierto, puesto que en las plataformas UNIX los sockets también se representan como descriptores de fichero. UDP no permite cerrar independientemente cada sentido de la comunicación (como sí puede hacer TCP), por lo que al cerrar un *socket* UDP ya no se pueden ni enviar ni recibir más datagramas.

El interfaz de programación de red de los Sistemas Operativos Windows se denomina Windows Sockets (WinSock)[1] e incluye la mayoría de las llamadas POSIX descritas anteriormente, aunque no es un API 100% compatible con el estándar POSIX y requiere cambios en el código fuente (por ejemplo en Windows los sockets no son descriptores de fichero, por lo que no se pueden utilizar funciones como close()). WinSock también define sus propias primitivas (WSA*), que incluyen funcionalidades adicionales específicas de Windows.

### 4.1.2. Utilización de 'strace' para analizar un cliente UDP

Como normalmente no se dispone del código fuente de las aplicaciones que se desean analizar (por eso se le denomina ingeniería inversa), vamos a comentar algunas herramientas que pueden ser útiles para analizar el comportamiento de red de las aplicaciones o comandos, más allá de las técnicas de desensamblado y depuración que se analizarán en profundidad en otras asignaturas de este Máster.

En los sistemas operativos UNIX como Linux existen varios comandos para trazar la ejecución de comandos, tales como 'ltrace' para monitorizar las llamadas a librerías y 'strace' para monitorizar las llamadas al sistema. Ésta última es muy útil por tanto para analizar el comportamiento de red de las aplicaciones, puesto que la mayoría de las funciones de red son llamadas al sistema. Además, aunque el código de la

[1] https://docs.microsoft.com/es-es/windows/win32/winsock/windows-sockets-start-page-2

aplicación esté ofuscado, las llamadas al sistema y sus parámetros no pueden estarlo, puesto que son interfaces de programación estándar del sistema huésped.

Como cualquier aplicación medianamente compleja puede realizar cientos o miles de llamadas de sistema, 'strace' permite filtrar el tipo de llamadas al sistema que se quieren trazar. En particular, para analizar el comportamiento de red de una aplicación se recomienda trazar el conjunto de llamadas al sistema de red (%net) y las relacionadas con los descriptores de fichero (%desc). Además, la opción '-tt' añade una marca de tiempo a cada llamada (con precisión de microsegundos), para poder saber cuándo se ejecutó cada una.

**Listado 4.5** – Traza de llamadas a sistema del cliente UDP con 'strace' (simplificado)

```
$ strace --trace=%net,%desc -tt -o udp_client.log ./udp_client localhost 7000
  "Hola localhost!"
15 bytes sent to UDP Server 127.0.0.1/7000
15 bytes received from UDP Server: "Hola localhost!"

$ cat udp_client.log
09:15:30.072314 openat(AT_FDCWD, "/etc/ld.so.cache", O_RDONLY|O_CLOEXEC) = 3
09:15:30.072438 fstat(3, {st_mode=S_IFREG|0644, st_size=69530, ...}) = 0
09:15:30.072461 mmap(NULL, 69530, PROT_READ, MAP_PRIVATE, 3, 0) =
            0x7f2cad56f000
09:15:30.072481 close(3)              = 0
09:15:30.072501 openat(AT_FDCWD, "/lib/x86_64-linux-gnu/libc.so.6", O_RDONLY|
            O_CLOEXEC) = 3
09:15:30.072522 read(3, "\177ELF\2\1\1\3\0\0\0\0\0\0\0\0\3\0>"..., 832) = 832
09:15:30.072540 pread64(3, "\6\0\0\0\4\0\0\0@\0\0\0\0\0\0"..., 784, 64) = 784
...
09:15:30.072594 fstat(3, {st_mode=S_IFREG|0755, st_size=2029224, ...}) = 0
09:15:30.072611 mmap(NULL, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|
                MAP_ANONYMOUS, -1, 0) = 0x7f2cad56d000
09:15:30.072632 pread64(3, "\6\0\0\0\4\0\0\0@\0\0\0\0\0\0"..., 784, 64) = 784
...
09:15:30.072685 mmap(NULL, 2036952, PROT_READ, MAP_PRIVATE|MAP_DENYWRITE, 3,
                0) = 0x7f2cad37b000
...
09:15:30.072802 close(3)              = 0


09:15:30.072955 socket(AF_NETLINK, SOCK_RAW|SOCK_CLOEXEC, NETLINK_ROUTE) = 3
09:15:30.072982 bind(3, {sa_family=AF_NETLINK, nl_pid=0, nl_groups=00000000},
                12) = 0
09:15:30.073003 getsockname(3, {sa_family=AF_NETLINK, nl_pid=22923,
                    nl_groups=00000000}, [12]) = 0
09:15:30.073028 sendto(3, {{len=20, type=RTM_GETADDR, flags=NLM_F_REQUEST|
                  NLM_F_DUMP, seq=1608676830, pid=0},
                  {ifa_family=AF_UNSPEC, ...}}, 20, 0,
                  {sa_family=AF_NETLINK, nl_pid=0, nl_groups=00000000},
                  12) = 20
09:15:30.073133 recvmsg(3, {msg_name={sa_family=AF_NETLINK, nl_pid=0,
                   nl_groups=00000000}, msg_namelen=12, msg_iov=[...],
```

```
                msg_iovlen=1, msg_controllen=0, msg_flags=0}, 0) = 164
...
09:15:30.073389 socket(AF_UNIX, SOCK_STREAM|SOCK_CLOEXEC|SOCK_NONBLOCK, 0) = 4
09:15:30.073410 connect(4, {sa_family=AF_UNIX,
                sun_path="/var/run/nscd/socket"}, 110) = -1 ENOENT (No
          existe el archivo o el directorio)
09:15:30.073533 close(4)            = 0
09:15:30.073552 close(3)            = 0
09:15:30.073573 socket(AF_UNIX, SOCK_STREAM|SOCK_CLOEXEC|SOCK_NONBLOCK, 0) = 3
09:15:30.073591 connect(3, {sa_family=AF_UNIX,
                sun_path="/var/run/nscd/socket"}, 110) = -1 ENOENT (No
          existe el archivo o el directorio)
09:15:30.073614 close(3)            = 0
09:15:30.073636 openat(AT_FDCWD, "/etc/nsswitch.conf", O_RDONLY|O_CLOEXEC) = 3
09:15:30.073669 fstat(3, {st_mode=S_IFREG|0644, st_size=542, ...}) = 0
09:15:30.073690 read(3, "# /etc/nsswitch.conf\n#\n# Example"..., 4096) = 542
09:15:30.073715 read(3, "", 4096)      = 0
09:15:30.073732 close(3)            = 0
09:15:30.073763 openat(AT_FDCWD, "/etc/host.conf", O_RDONLY|O_CLOEXEC) = 3
09:15:30.073784 fstat(3, {st_mode=S_IFREG|0644, st_size=92, ...}) = 0
09:15:30.073804 read(3, "# The \"order\" line is only used "..., 4096) = 92
09:15:30.073823 read(3, "", 4096)      = 0
09:15:30.073839 close(3)            = 0
09:15:30.073855 openat(AT_FDCWD, "/etc/resolv.conf", O_RDONLY|O_CLOEXEC) = 3
09:15:30.073876 fstat(3, {st_mode=S_IFREG|0644, st_size=717, ...}) = 0
09:15:30.073894 read(3, "# This file is managed by man:sy"..., 4096) = 717
09:15:30.073914 read(3, "", 4096)      = 0
09:15:30.073930 close(3)            = 0
09:15:30.073960 openat(AT_FDCWD, "/etc/ld.so.cache", O_RDONLY|O_CLOEXEC) = 3
09:15:30.073980 fstat(3, {st_mode=S_IFREG|0644, st_size=69530, ...}) = 0
09:15:30.073998 mmap(NULL, 69530, PROT_READ, MAP_PRIVATE, 3, 0) =
          0x7f2cad56f000
09:15:30.074016 close(3)            = 0
09:15:30.074034 openat(AT_FDCWD, "/lib/x86_64-linux-gnu/libnss_files.so.2",
               O_RDONLY|O_CLOEXEC) = 3
09:15:30.074054 read(3, "\177ELF\2\1\1\0\0\0\0\0\0\0\0\0\3\0>"..., 832) = 832
09:15:30.074073 fstat(3, {st_mode=S_IFREG|0644, st_size=51832, ...}) = 0
09:15:30.074091 mmap(NULL, 79672, PROT_READ, MAP_PRIVATE|MAP_DENYWRITE, 3, 0)
          = 0x7f2cad367000
...
09:15:30.074226 close(3)            = 0
09:15:30.074296 openat(AT_FDCWD, "/etc/hosts", O_RDONLY|O_CLOEXEC) = 3
09:15:30.074325 lseek(3, 0, SEEK_CUR)   = 0
09:15:30.074342 fstat(3, {st_mode=S_IFREG|0644, st_size=229, ...}) = 0
09:15:30.074360 read(3, "127.0.0.1\tlocalhost\n127.0.1.1\tub"..., 4096) = 229
09:15:30.074380 lseek(3, 0, SEEK_CUR)   = 229
09:15:30.074400 read(3, "", 4096)      = 0
09:15:30.074416 close(3)            = 0


09:15:30.074433 socket(AF_INET, SOCK_DGRAM, IPPROTO_UDP) = 3
09:15:30.074453 connect(3, {sa_family=AF_INET, sin_port=htons(7000),
                sin_addr=inet_addr("127.0.0.1")}, 16) = 0
09:15:30.074477 getpeername(3, {sa_family=AF_INET, sin_port=htons(7000),
```

```
                    sin_addr=inet_addr("127.0.0.1")}, [128->16]) = 0
09:15:30.074501 sendto(3, "Hola localhost!", 15, 0, NULL, 0) = 15
09:15:30.074634 fstat(1, {st_mode=S_IFCHR|0620, st_rdev=makedev(0x88, 0x3),
                ...}) = 0
09:15:30.074658 write(1, "15 bytes sent to UDP Server 127."..., 43) = 43
09:15:30.074743 poll([{fd=3, events=POLLIN|POLLERR}], 1, 3000) = 1 ([{fd=3,
            revents=POLLIN}])
09:15:30.074768 recvfrom(3, "Hola localhost!", 1023, 0, NULL, NULL) = 15
09:15:30.074793 write(1, "15 bytes received from UDP Serve"..., 53) = 53
09:15:30.074865 close(3)            = 0
09:15:30.074935 +++ exited with 0 +++
```

A pesar de que al principio del **Listado 4.5** aparecen un gran número de llamadas al sistema relacionadas con la propia ejecución del programa, en la parte final (a partir de 09:15:30.074433) se pueden observar claramente todas las llamadas al sistema (marcadas en negrita) que aparecen en el código fuente del **Listado 4.1**, aunque con ciertos matices. Por ejemplo, a pesar de que el programa utiliza las funciones send() y recv(), las llamadas al sistema que realmente se ejecutan son sendto() y recvfrom(), aunque sin especificar las direcciones remotas. Tampoco aparecen las llamadas getaddrinfo() y getnameinfo(), pero sí sus efectos. Por ejemplo, los sockets AF_NETLINK y AF_UNIX se utilizan para comunicarse con el núcleo y con otros procesos del sistema, para saber qué direcciones IP están configuradas y convertir el nombre "localhost" a la dirección IPv4 de *loopback* 127.0.0.1 (e.g. openat(AT_FDCWD, "/etc/hosts", ...)).

En Windows no existe un comando como 'strace', pero se puede utilizar 'Event Tracing for Windows (ETW)' para registrar los eventos de WinSock[2] para analizarlos posteriormente en el visor de eventos. Para un análisis más informal de la actividad de una aplicación Windows (aunque no permite llegar al detalle de 'strace' o de ETW), también se puede utilizar la herramienta 'Process Monitor'[3] (ver **Figura 4.6**) que forma parte del paquete 'Sysinternals'.

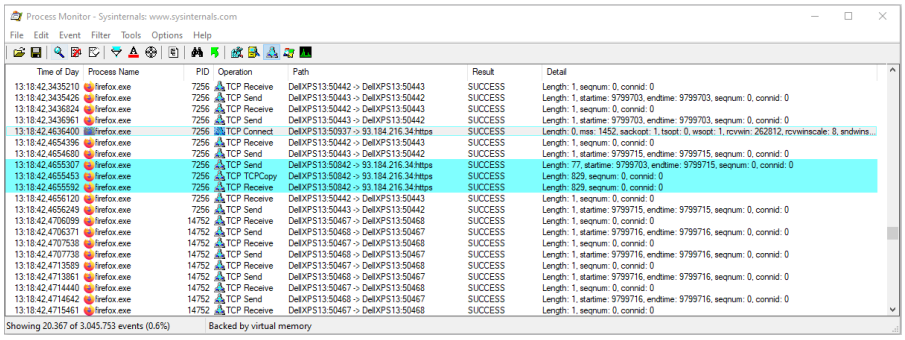



**Figura 4.6** – La herramienta 'Process Monitor' de Windows.

[2] https://docs.microsoft.com/en-us/windows/win32/winsock/winsock-tracing
[3] https://docs.microsoft.com/es-es/sysinternals/downloads/procmon

## 4.2. Análisis de una aplicación servidor UDP

Analicemos ahora una aplicación UDP de tipo servidor, que recibe peticiones de diferentes clientes y les envía respuesta. En este caso devuelve simplemente el mensaje que le envía el cliente (e.g. servidor *echo*). Obviamente los servidores UDP son bastante más complejos, pero este ejemplo sirve para ilustrar la estructura que debería tener cualquier servidor UDP.

**Listado 4.7** – Código fuente del servidor UDP de ejemplo (udp_server.c)

```
#define BUFFER_SIZE 1024

int main ( int argc, char* argv[] )
{
 if (argc < 2 || argc > 3) {
  const char * myself = basename(argv[0]);
  fprintf(stderr, "Usage: %s <port> [<addr>]\n", myself);
  fprintf(stderr, "   <port>: Port number to bind to\n");
  fprintf(stderr, "   <addr>: IP Address to bind to\n");
  exit(EXIT_FAILURE);
 }
 const char* port_arg = argv[1];
 const char* addr_arg;
 if (argc == 3) {
  addr_arg = argv[2];
 } else {
  addr_arg = NULL;
 }

 /* Obtain server address(es) matching host/port */
 struct addrinfo hints;
 memset(&hints, 0x00, sizeof(hints));
 hints.ai_family = AF_UNSPEC;     /* Allow IPv4 or IPv6 addresses */
 hints.ai_socktype = SOCK_DGRAM;  /* Datagram socket */
 hints.ai_flags = AI_PASSIVE | AI_NUMERICHOST | AI_ADDRCONFIG;
                  /* Ask for a configured server address */
 hints.ai_protocol = IPPROTO_UDP; /* UDP protocol */
 struct addrinfo *result;
 int err = getaddrinfo(addr_arg, port_arg, &hints, &result);
 if (err != 0) {
  fprintf(stderr, "getaddrinfo() error: %s\n", gai_strerror(err));
  exit(EXIT_FAILURE);
 }

 /* getaddrinfo() returns a list of address structures. Try each
  * address until we successfullty bind(). If socket() or bind()
  * fails, we close the socket and try the next address. */
 int sock_fd;
 struct addrinfo *ptr;
```

```
for (ptr=result; ptr!=NULL; ptr=ptr->ai_next) {

  /* Open an UDP socket */
  sock_fd = socket(ptr->ai_family, ptr->ai_socktype,
                   ptr->ai_protocol);
  if (sock_fd == -1) {
    continue; /* Failure! Try next address */
  }


  /* Bind to UDP port */
  err = bind(sock_fd, ptr->ai_addr, ptr->ai_addrlen);
  if (err == 0) {
    break; /* Success! No need to keep looking for more addresses */

  }

  close(sock_fd); /* Close socket because bind() failed */
}

if (ptr == NULL) {
  /* No address succeeded */
  fprintf(stderr, "Cannot bind to UDP port '%s'\n", port_arg);
  exit(EXIT_FAILURE);
}

freeaddrinfo(result); /* No longer needed */


/* Get Server IP address and port */
struct sockaddr_storage server_sa;
socklen_t server_sa_len = sizeof(server_sa);
err = getsockname(sock_fd, (struct sockaddr *) &server_sa,
                  &server_sa_len);
if (err == -1) {
  perror("getsockname() error");
  exit(EXIT_FAILURE);
}

char server_addr_str[NI_MAXHOST];
char server_port_str[NI_MAXSERV];
int gni_flags = NI_NUMERICHOST | NI_NUMERICSERV;
err = getnameinfo((struct sockaddr *) &server_sa, server_sa_len,
                  server_addr_str, sizeof(server_addr_str),
                  server_port_str, sizeof(server_port_str),
                  gni_flags);
if (err == -1) {
  fprintf(stderr, "getnameinfo() error: %s\n", gai_strerror(err));
```

```
  exit(EXIT_FAILURE);
}
printf("UDP Server receiving at %s/%s ...\n",
      server_addr_str, server_port_str);
printf("Press Ctrl+C to exit.\n");

char buffer[BUFFER_SIZE];
struct sockaddr_storage client_sa;
socklen_t client_sa_len = sizeof(client_sa);

char client_addr_str[NI_MAXHOST];
char client_port_str[NI_MAXSERV];

for(;;) {

  /* Wait until we receive some data from a client */
  ssize_t bytes_recv = recvfrom(sock_fd, buffer, sizeof(buffer), 0,
                                (struct sockaddr*) &client_sa,
                        &client_sa_len);
  if (bytes_recv == -1) {
    perror("recv() error");
    break;
  }

  /* Get client IP address and port */
  err = getnameinfo((struct sockaddr *) &client_sa, client_sa_len,
                    client_addr_str, sizeof(client_addr_str),
                    client_port_str, sizeof(client_port_str),
              gni_flags);
  if (err == -1) {
    fprintf(stderr, "getnameinfo() error: %s\n", gai_strerror(err));
    break;
  }
  buffer[bytes_recv] = '\0';
  printf("%ld bytes received from UDP Client %s/%s: \"%s\"\n",
              bytes_recv, client_addr_str, client_port_str, buffer);


  /* Send a response back to the client */
  ssize_t bytes_sent = sendto(sock_fd, buffer, bytes_recv, 0,
                              (struct sockaddr*) &client_sa,
                        client_sa_len);
  if (bytes_sent == -1) {
    perror("sendto() error");
    break;
  }

  printf("%ld bytes sent to UDP Client %s/%s: \"%s\"\n",
              bytes_sent, client_addr_str, client_port_str, buffer);
```

```
  }

  /* Close socket and exit */
  err = close(sock_fd);
  if (err != 0) {
    perror("close() error");
  }

  return EXIT_FAILURE;
}
```

**Listado 4.8** – Información de uso del servidor UDP de ejemplo (udp_server.c)

```
$ ./udp_server
Usage: udp_server <port> [<addr>]
  <port>: Port number to bind to
  <addr>: IP Address to bind to
```

**Listado 4.9** – Ejecución del servidor UDP de ejemplo (udp_server)

```
$ ./udp_server 7000 ::
UDP Server receiving at ::/7000 ...
Press Ctrl+C to exit.
15 bytes received from UDP Client ::ffff:127.0.0.1/41219: "Hola localhost!"
15 bytes sent to UDP Client ::ffff:127.0.0.1/41219: "Hola localhost!"
9 bytes received from UDP Client ::1/58535: "Hola ::1!"
9 bytes sent to UDP Client ::1/58535: "Hola ::1!"
18 bytes received from UDP Client ::ffff:192.168.1.110/32834: "Hola 192.168.1.33!"
18 bytes sent to UDP Client ::ffff:192.168.1.110/32834: "Hola 192.168.1.33!"
^C
```

**Listado 4.10** – Información del comando 'netstat' sobre servidores UDP

```
$ netstat -ulnp
Conexiones activas de Internet (servidores y establecidos)
Proto Recib Enviad Dir. local    Dir. Remota  Estado  PID/Program
udp       0      0 127.0.0.53:53 0.0.0.0:*           -
udp6      0      0 :::7000       :::*                2561/udp_server
```

**Listado 4.11** – Información del comando 'nmap' sobre los puertos UDP abiertos

```
$ sudo nmap -sU -n -p1-49151 192.168.1.33
Starting Nmap 7.80 ( https://nmap.org )
Nmap scan report for 192.168.1.33
Host is up (0.0000030s latency).
Not shown: 49150 closed ports
PORT     STATE        SERVICE
7000/udp open         afs3-fileserver

Nmap done: 1 IP address (1 host up) scanned in 1.76 seconds
```

Además de cambio de orden entre las operaciones de recepción y envío respecto al cliente (ver **Listado 4.1**) y de que éstas se encuentran en un bucle infinito, la estructura del programa es bastante similar. Las principales diferencias consisten en la utilización de la llamada bind() para elegir la dirección y el puerto UDP en el que se desea escuchar, y en la utilización de recvfrom() y sendto() por el servidor en lugar de las llamadas connect(), send() y recv() que utiliza el cliente, porque en este caso el servidor puede enviar y recibir mensajes a muchos destinos y orígenes diferentes.

El **Listado 4.9** muestra la ventaja de implementar código agnóstico de la versión del protocolo IP subyacente (gracias a la utilización de las funciones getaddrinfo()y getnameinfo()), puesto que el servidor es capaz de recibir peticiones tanto como IPv6 (::1) como IPv4 (192.168.1.10), aunque, como el servidor UDP se ha asociado a una dirección de transporte IPv6 (::), recibe las direcciones IPv4 (32 bits) mapeadas dentro de direcciones IPv6 (128 bits), como, por ejemplo ::ffff:192.168.1.20.

Otra forma de ver en qué puerto está escuchando un servidor UDP es mediante el comando 'netstat' (que está disponible en la mayoría de los sistemas operativos, como Linux, Windows, macOS), que muestra información sobre los procesos que están a la escucha o tienen conexiones de red activas. Por ejemplo, el **Listado 4.10** muestra todos los servidores UDP (opción -u) que están a la escucha (opción -l), así como el proceso (opción -p) asociado, aunque esta última opción solo muestra información sobre los procesos del usuario. Para ver la información de cualquier proceso es necesario ejecutarlo como superusuario.

Si no es posible acceder al sistema origen para ejecutar el comando 'netstat' una alternativa para intentar detectar que puertos UDP están abiertos (i.e. tienen procesos a la escucha), es el comando 'nmap'. El **Listado 4.11** muestra un escaneo de todos los puertos UDP (opción -sU) bien conocidos y reservados (opción -p 1-49151) realizado con 'nmap'.

### 4.2.1. Principales llamadas al sistema de un servidor UDP

A continuación, se describen las operaciones más importantes del servidor. Algunas de ellas ya se describieron en el apartado del cliente UDP, así que no se repetirán aquí.

- **int getaddrinfo(const char *node, const char *service, const struct addrinfo *hints, struct addrinfo **res)** – Esta función ya se ha comentado para el cliente UDP, así que solo se mencionará que en este caso se utilizan los *flags* AI_PASSIVE | AI_NUMERICHOST | AI_ADDRCONFIG para, respectivamente, devolver solo direcciones que sean susceptibles de hacer un bind() posterior, deben ser numéricas para evitar una resolución de nombres, y comprobar que las direcciones devueltas son compatibles con las direcciones configuradas en los interfaces de red del equipo (e.g. para evitar que se devuelvan direcciones IPv6 globales si no hay ninguna configurada).

- **int socket(int domain, int type, int protocol)** – Aunque esta función ya se describió en el cliente UDP, cabe destacar que el servidor utiliza exactamente los mismos parámetros para crear un *socket* UDP, puesto que lo que realmente indica si un socket va a ser empleado como cliente o servidor es la llamada a bind().

- **int bind(int sockfd, const struct sockaddr *addr, socklen_t addrlen)** – Esta función le indica al sistema la dirección de transporte que se desea utilizar, por lo que normalmente solo la realizan los servidores, que deben utilizar una dirección y puerto conocida por el cliente, mientras que los clientes pueden utilizar cualquier dirección de transporte disponible en el equipo, puesto que a los servidores les basta con responder a la dirección y puerto origen de las peticiones que reciben. Es bastante habitual que solo se indique un número de puerto, y en ese caso se utiliza por defecto la denominada dirección comodín (*wildcard* en inglés), qué es 0.0.0.0 (32 bits a cero) para los *sockets* IPv4 y :: (128 bits a cero) para los *sockets* IPv6, y que indica al sistema que el servidor puede recibir peticiones de cualquier dirección IP configurada en el equipo. Si se define una dirección IP explícitamente diferente a la dirección comodín, el servidor solo recibirá peticiones enviadas a esa dirección IP. Por ejemplo si se utilizan las direcciones de *loopback* de IPv4 (127.0.0.1) o IPv6 (::1), el servidor solo estará accesible para los clientes que ejecuten en la misma máquina, pero no para clientes remotos. En este caso se utiliza el puerto 7000, que está en el rango de los puertos registrados (1.024-49.151), para que no sea necesario ejecutar el servidor UDP con permisos de superusuario.

- **int getsockname(int sockfd, const struct sockaddr *addr, socklen_t addrlen)** – Esta es la función complementaria a getpeername() que se utilizaba en el cliente UDP, puesto que getsockname() devuelve la dirección de transporte del propio *socket* abierto localmente, por ejemplo para comprobar si la función bind() ha funcionado correctamente o para conocer el puerto efímero (49.151-6.5535) empleado por un cliente. Como en el caso de getpeername(), getsockname() devuelve la dirección de transporte en formato binario, por lo que se quiere imprimir se debe emplear getnameinfo(), que permite procesar tanto direcciones IPv4 como IPv6, y que ya se describió en el apartado sobre el cliente UDP.

- **ssize_t recvfrom(int sockfd, void *buf, size_t len, int flags, struct sockaddr *src_addr, socklen_t *addrlen)** – Esta función permite recibir mensajes UDP, almacenándolos en el *buffer* especificado, siempre que su tamaño no exceda del tamaño del *buffer*. Pero, a diferencia de recv() que utilizaba el cliente UDP, además devuelve la dirección y puerto origen que envió el mensaje, por lo permite a los servidores conocer la dirección de los clientes para enviarles una respuesta mediante sendto(). En este caso no se emplea poll() o select()para implementar un temporizador antes de recibir datos, como hacía el cliente UDP, puesto que el servidor puede recibir mensajes de algún cliente en cualquier momento, y se delega en el cliente la responsabilidad de reenviar la petición si no recibe respuesta. Este comportamiento sin retransmisiones es típico de los servidores UDP como DNS o NTP, aunque también existen protocolos basados en UDP que sí implementan mecanismos de detección y recuperación frente a errores, y que por tanto sí requieren la utilización de temporizadores. Sin embargo, para el envío de grandes cantidades de información se recomienda el uso de un protocolo orientado a conexión y fiable como TCP.

- **ssize_t sendto(int sockfd, const void *buf, size_t len, int flags, const struct sockaddr *dest_addr, socklen_t addrlen)** – Esta función, como el send() del cliente UDP, permite enviar un mensaje UDP a través de un *socket*, aunque en este caso no es necesario asociar el *socket* UDP a un destino con connect(), sino que cada mensaje se puede enviar un destino diferente, lo que lo hace ideal para servidores UDP que envían una única respuesta por petición cliente, como este caso.

- **int close(int fd)** – Normalmente los servidores nunca cierran el *socket* desde el reciben las peticiones de los clientes a no ser que reciban una señal para salir o, como en este caso, si algún error del sistema hace salir al flujo del bucle infinito. Normalmente el sistema operativo se encarga de liberar los recursos

(e.g. *sockets*) asignados a un proceso que termina automáticamente, aunque en algunos casos es mejor cerrarlos explícitamente, por ejemplo, para que se terminen de enviar datos pendientes.

### 4.2.2. Utilización de 'strace' para analizar un servidor UDP

Una vez entendido el código fuente del servidor UDP de ejemplo, vamos a estudiar su traza de llamadas al sistema, obtenidas mediante el comando 'strace', que ya se describió en el mismo apartado del cliente UDP.

**Listado 4.12** – Traza de llamadas a sistema del servidor UDP con 'strace' (simplificado)

```
$ strace --trace=%net,%desc -tt -o udp_server.log ./udp_server 7000 ::
UDP Server receiving at ::/7000 ...
Press Ctrl+C to exit.
15 bytes received from UDP Client ::ffff:127.0.0.1/56358: "Hola localhost!"
15 bytes sent to UDP Client ::ffff:127.0.0.1/56358: "Hola localhost!"
9 bytes received from UDP Client ::1/50921: "Hola ::1!"
9 bytes sent to UDP Client ::1/50921: "Hola ::1!"
^C

$ cat udp_server.log
09:30:37.366566 openat(AT_FDCWD, "/etc/ld.so.cache", O_RDONLY|O_CLOEXEC) = 3
09:30:37.366644 fstat(3, {st_mode=S_IFREG|0644, st_size=69530, ...}) = 0
09:30:37.366668 mmap(NULL, 69530, PROT_READ, MAP_PRIVATE, 3, 0) =
            0x7f6f7e62f000
09:30:37.366689 close(3)            = 0
09:30:37.366710 openat(AT_FDCWD, "/lib/x86_64-linux-gnu/libc.so.6", O_RDONLY|
                O_CLOEXEC) = 3
09:30:37.366732 read(3, "\177ELF\2\1\1\3\0\0\0\0\0\0\0\0\3\0>"..., 832) = 832
09:30:37.366751 pread64(3, "\6\0\0\0\4\0\0\0@\0\0\0\0\0\0\0"..., 784, 64) = 784
...
09:30:37.366837 fstat(3, {st_mode=S_IFREG|0755, st_size=2029224, ...}) = 0
09:30:37.366857 mmap(NULL, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|
                MAP_ANONYMOUS, -1, 0) = 0x7f6f7e62d000
09:30:37.366891 pread64(3, "\6\0\0\0\4\0\0\0@\0\0\0\0\0\0"..., 784, 64) = 784
...

09:30:37.366949 mmap(NULL, 2036952, PROT_READ, MAP_PRIVATE|MAP_DENYWRITE, 3,
                0) = 0x7f6f7e43b000
...
09:30:37.367074 close(3)            = 0


09:30:37.367236 socket(AF_NETLINK, SOCK_RAW|SOCK_CLOEXEC, NETLINK_ROUTE) = 3
09:30:37.367263 bind(3, {sa_family=AF_NETLINK, nl_pid=0, nl_groups=00000000},
                12) = 0
09:30:37.367287 getsockname(3, {sa_family=AF_NETLINK, nl_pid=22861,
```

```
                      nl_groups=00000000}, [12]) = 0
09:30:37.367314 sendto(3, {{len=20, type=RTM_GETADDR, flags=NLM_F_REQUEST|
                    NLM_F_DUMP, seq=1608675157, pid=0},
                    {ifa_family=AF_UNSPEC, ...}}, 20, 0,
                    {sa_family=AF_NETLINK, nl_pid=0, nl_groups=00000000},
                    12) = 20
09:30:37.367430 recvmsg(3, {msg_name={sa_family=AF_NETLINK, nl_pid=0,
                     nl_groups=00000000}, msg_namelen=12, msg_iov=[...],
                     msg_iovlen=1, msg_controllen=0, msg_flags=0}, 0) = 164
...
09:30:37.367688 socket(AF_UNIX, SOCK_STREAM|SOCK_CLOEXEC|SOCK_NONBLOCK, 0) = 4
09:30:37.367708 connect(4, {sa_family=AF_UNIX,
                    sun_path="/var/run/nscd/socket"}, 110) = -1 ENOENT
              (No existe el archivo o el directorio)
09:30:37.367859 close(4)              = 0
09:30:37.367881 close(3)              = 0


09:30:37.367905 socket(AF_INET6, SOCK_DGRAM, IPPROTO_UDP) = 3
09:30:37.367926 bind(3, {sa_family=AF_INET6, sin6_port=htons(7000),
                  sin6_flowinfo=htonl(0), inet_pton(AF_INET6, "::",
                  &sin6_addr), sin6_scope_id=0}, 28) = 0
09:30:37.367950 getsockname(3, {sa_family=AF_INET6, sin6_port=htons(7000),
                     sin6_flowinfo=htonl(0), inet_pton(AF_INET6, "::",
                     &sin6_addr), sin6_scope_id=0}, [128->28]) = 0
09:30:37.367981 fstat(1, {st_mode=S_IFCHR|0620, st_rdev=makedev(0x88, 0x4),
                  ...}) = 0
09:30:37.368003 write(1, "UDP Server receiving at ::/7000 "..., 36) = 36
09:30:37.368096 write(1, "Press Ctrl+C to exit.\n", 22) = 22

09:30:37.368165 recvfrom(3, "Hola localhost!", 1024, 0, {sa_family=AF_INET6,
                     sin6_port=htons(56358), sin6_flowinfo=htonl(0),
                     inet_pton(AF_INET6, "::ffff:127.0.0.1", &sin6_addr),
                     sin6_scope_id=0}, [128->28]) = 15
09:30:41.869017 write(1, "15 bytes received from UDP Clien"..., 76) = 76
09:30:41.869127 sendto(3, "Hola localhost!", 15, 0, {sa_family=AF_INET6,
                    sin6_port=htons(56358), sin6_flowinfo=htonl(0),
                    inet_pton(AF_INET6, "::ffff:127.0.0.1", &sin6_addr),
                    sin6_scope_id=0}, 28) = 15
09:30:41.869168 write(1, "15 bytes sent to UDP Client ::ff"..., 70) = 70

09:30:41.869532 recvfrom(3, "Hola ::1!", 1024, 0, {sa_family=AF_INET6,
                     sin6_port=htons(50921), sin6_flowinfo=htonl(0),
                     inet_pton(AF_INET6, "::1", &sin6_addr),
                     sin6_scope_id=0}, [28]) = 9
09:30:43.861380 write(1, "9 bytes received from UDP Client"..., 56) = 56
09:30:43.861488 sendto(3, "Hola ::1!", 9, 0, {sa_family=AF_INET6,
                    sin6_port=htons(50921), sin6_flowinfo=htonl(0),
                    inet_pton(AF_INET6, "::1", &sin6_addr),
                    sin6_scope_id=0}, 28) = 9
09:30:43.861548 write(1, "9 bytes sent to UDP Client ::1/5"..., 50) = 50

09:30:43.861639 recvfrom(3, 0x7fffb4db1fa0, 1024, 0, 0x7fffb4db1ee0, [28]) = ?
              ERESTARTSYS (To be restarted if SA_RESTART is set)
```

```
09:30:47.268622 --- SIGINT {si_signo=SIGINT, si_code=SI_KERNEL} ---
09:30:47.268779 +++ killed by SIGINT +++
```

En el **Listado 4.12**, además de las llamadas al sistema para la carga de las librerías dinámicas del ejecutable, y de las llamadas al núcleo y a otros procesos (*sockets* AF_NETLINK y AF_UNIX) para resolver los parámetros de getaddrinfo() (que es un proceso bastante más corto que en el caso del cliente UDP puesto que ahora no es necesario resolver un nombre del dominio), a partir del instante 09:30:37.367905 se pueden observar la mayoría de las operaciones de red que se describen en el código fuente del servidor UDP (**Listado 4.9**). En particular, los saltos en las marcas de tiempo permiten distinguir claramente la creación del *socket* UDP (AF_INET6 debido al uso de la dirección comodín IPv6 '::') y los dos mensajes que se reciben de los clientes (recvfrom()) y las respuestas que se envían de vuelta (sendto()) a las mismas direcciones de transporte. Solo comentar que como 'strace' establece la marca de tiempo cuando se realiza la llamada al sistema, y como recvfrom() es una llamada bloqueante, la diferencia de tiempo entre recvfrom() y la siguiente llamada al sistema realmente es el tiempo que ha estado esperando para recibir un mensaje. La opción -T de 'strace' muestra el tiempo empleado por cada llamada al sistema.

El comando 'strace' también permite trazar las señales que recibe el proceso siendo monitorizado, por lo que al final de listado se puede ver la señal SIGINT generada al pulsar Ctrl+C para cerrar el servidor UDP (que de otro modo se mantendría en un bucle infinito).

## 4.3. Análisis del tráfico de una aplicación cliente-servidor UDP

Una vez analizadas las aplicaciones cliente y servidor UDP por separado, vamos a analizar el comportamiento completo de la aplicación y de su protocolo.

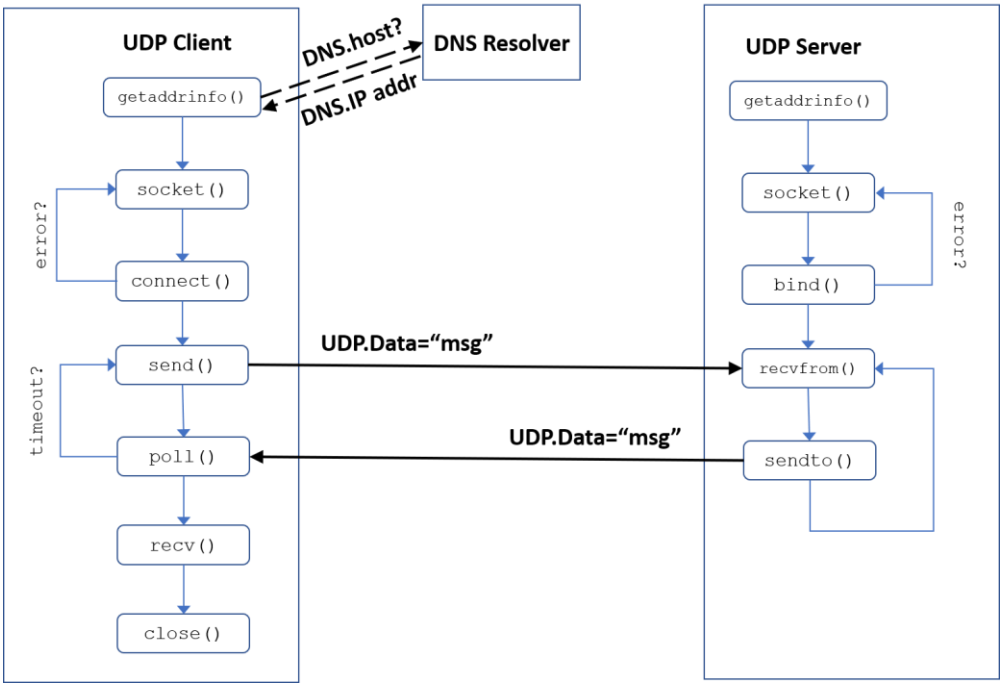



**Figura 4.13** – Modelo de la aplicación cliente-servidor UDP de ejemplo

La **Figura 4.13** muestra un modelo simplificado del flujo de ejecución de las aplicaciones cliente y servidor, así como el tráfico que envía cada una de ellas.

Aunque no todavía no se ha visto en ninguna de las trazas de llamadas al sistema que hemos analizado hasta ahora, la llamada getaddrinfo() puede generar tráfico de red si al cliente se le pasa como argumento el nombre DNS del servidor, en cuyo caso debe consultar contra el servidor DNS recursivo que tenga configurado el equipo.

En cuanto al tráfico propiamente dicho de la aplicación, consiste únicamente en un mensaje de texto que se envía del cliente al servidor en un datagrama UDP, y el mismo mensaje en sentido contrario. En caso de que el servidor no responda en menos de 3 segundos (bien porque no esté disponible o porque se pierda el mensaje de ida o de vuelta), el cliente puede retransmitir hasta 3 veces su mensaje.

Nótese que, aunque este la implementación de un *timeout* es el comportamiento recomendado para un protocolo no fiable como UDP, puede haber aplicaciones que solo envíen un único mensaje UDP y se queden esperando respuesta (en cuyo caso no habría llamada poll() y la flecha de sendto() iría contra recv()). De mismo modo, aunque el uso recomendado de getaddrinfo() es probar con todas las direcciones que devuelve (socket() + connect() o socket() + bind()), puede haber aplicaciones que solo intenten usar la primera dirección devuelta. Del mismo modo, nada impide implementar el cliente sin connect() y con llamadas a sendto() y recvfrom(), aunque en ese caso se debería comprobar que la respuesta viene del mismo servidor al que se envió el mensaje (connect() se encarga de eso). Por tanto, hay que entender este

diagrama como un diseño habitual de las aplicaciones UDP, pero de ningún modo es la única implementación posible.

**Listado 4.14** – Configuración y ejecuciones del cliente UDP en el equipo 192.168.1.33

```
muruenya@192.168.1.33:~$ route -n
Tabla de rutas IP del núcleo
Destino        Pasarela       Genmask        Indic Métric Ref    Uso Interfaz
0.0.0.0        192.168.1.1    0.0.0.0        UG    100    0        0 eth0
192.168.1.0    0.0.0.0        255.255.255.0  U     100    0        0 eth0

muruenya@192.168.1.33:~$ cat /etc/hosts
127.0.0.1           localhost
::1         ip6-localhost ip6-loopback

muruenya@192.168.1.33:~$ cat /etc/resolv.conf
nameserver 127.0.0.53

muruenya@192.168.1.33:~$ resolvectl dns eth0
Link 2 (eth0): 80.58.61.250 80.58.61.254

muruenya@192.168.1.33:~$ ./udp_client www.example.com 7000 "Hola  www.example.com!"
21 bytes sent to UDP Server 93.184.216.34/7000
Timeout! Retrying #1
21 bytes sent to UDP Server 93.184.216.34/7000
Timeout! Retrying #2
21 bytes sent to UDP Server 93.184.216.34/7000
Cannot contact UDP Server 93.184.216.34/7000

muruenya@192.168.1.33:~$ ./udp_client localhost 7000 "Hola localhost!"
15 bytes sent to UDP Server 127.0.0.1/7000
15 bytes received from UDP Server: "Hola localhost!"

muruenya@192.168.1.33:~$ ./udp_client ::1 7000 "Hola ::1!"
9 bytes sent to UDP Server ::1/7000
9 bytes received from UDP Server: "Hola ::1!"
```

**Listado 4.15** – Configuración y ejecución del cliente UDP en el equipo 192.168.1.110

```
muruenya@192.168.1.110:~$ route -n
Kernel IP routing table
Destination     Gateway         Genmask         Flags Metric Ref    Use Iface
0.0.0.0         192.168.1.1     255.255.255.255 U     0      0        0 eth2
192.168.1.110   0.0.0.0         255.255.255.255 U     256    0        0 eth2

muruenya@192.168.1.110:~$ cat /etc/resolv.conf
nameserver 80.58.61.250
nameserver 80.58.61.254

muruenya@192.168.1.110:~$ ./udp_client 192.168.1.33 7000 "Hola 192.168.1.33!"
```

```
18 bytes sent to UDP Server 192.168.1.33/7000
18 bytes received from UDP Server: "Hola 192.168.1.33!"
```

**Listado 4.16** – Ejecución del servidor UDP de ejemplo en el equipo 192.168.1.33

```
muruenya@192.168.1.33:~$ ./udp_server 7000 ::
UDP Server receiving at ::/7000 ...
Press Ctrl+C to exit.
15 bytes received from UDP Client ::ffff:127.0.0.1/48289: "Hola localhost!"
15 bytes sent to UDP Client ::ffff:127.0.0.1/48289: "Hola localhost!"
9 bytes received from UDP Client ::1/40285: "Hola ::1!"
9 bytes sent to UDP Client ::1/40285: "Hola ::1!"
18 bytes received from UDP Client ::ffff:192.168.1.110/55745: "Hola 192.168.1.33!"
18 bytes sent to UDP Client ::ffff:192.168.1.110/55745: "Hola 192.168.1.33!"
```

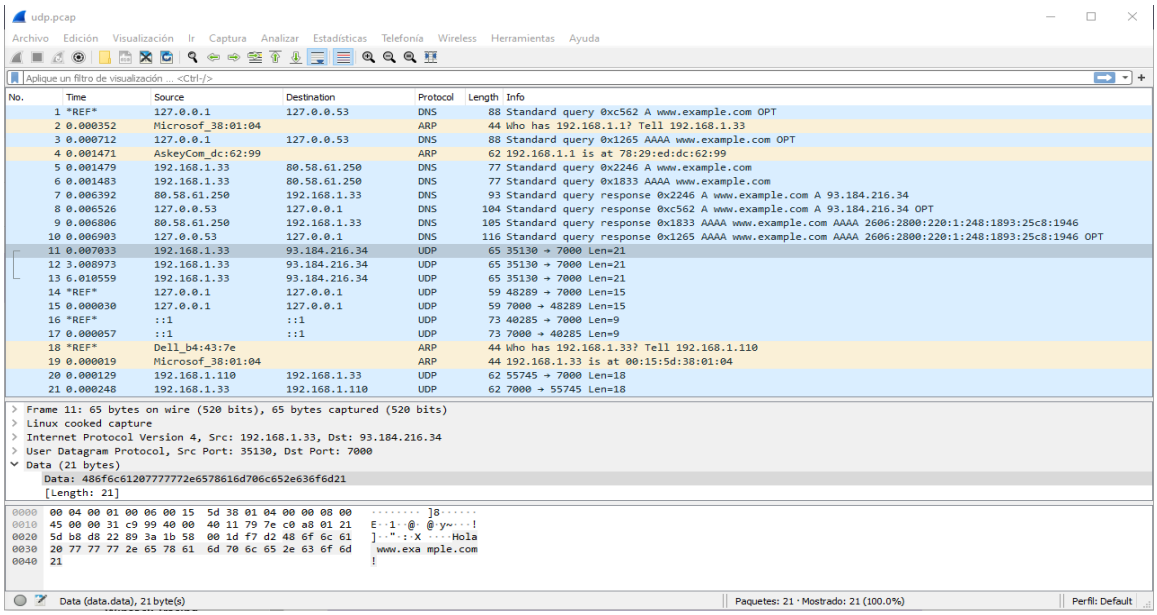



**Figura 4.17** – Captura de tráfico de la aplicación cliente-servidor UDP de ejemplo (udp.pcap)

Pasemos ahora a analizar el tráfico de red generado por esta aplicación. La **Figura 4.17** muestra una captura de tráfico de red realizada desde el equipo 192.168.1.33 (sudo tcpdump -i any -s0 -w udp.pcap), donde está ejecutando el servidor UDP (ver **Listado 4.16**) y adicionalmente un cliente UDP, que envía consultas tanto al servidor local como al servidor www.example.com (ver **Listado 4.14**). Adicionalmente, para ver tráfico externo contra el servidor UDP, también se ha ejecutado el cliente UDP desde el equipo 192.168.1.110 (ver **Listado 4.15**), que está en la misma subred (192.168.1.0/24). Para distinguir cada una de las peticiones, se ha establecido una marca de tiempo en el primer paquete generado por cada una de ellas.

Nótese que la única forma de capturar tráfico en el interfaz de *loopback* de un equipo es realizando la captura en el propio equipo, porque ese tráfico nunca pasa por un interfaz de red físico. Por lo tanto, incluso si hubiésemos realizado la captura con un *tap* o con *port mirroring* no habríamos visto las dos peticiones del cliente interno a localhost/127.0.0.1 (paquetes #14-15) y ::1 (paquetes #16-17).

Pero empecemos analizando la primera petición que realiza el cliente UDP en 192.168.1.33, y que, en lugar de ir contra el servidor UDP local, intenta comunicarse con el servidor www.example.com. Aunque el primer mensaje enviado por el cliente UDP propiamente dicho sería el paquete seleccionado (#11), realmente todos los paquetes anteriores también se deben al cliente, ya que a getaddrinfo() se le ha pasado un nombre DNS (www.example.com). En particular, como a getaddrinfo() se le pasa como familia de protocolos AF_UNSPEC, intenta obtener tanto direcciones IPv4 (registro DNS tipo A) como IPv6 (registro DNS tipo AAAA). Sin embargo, aunque las pilas modernas normalmente priorizan IPv6 frente a IPv4, como el equipo local no tiene configurada ninguna dirección IPv6 global, getaddrinfo() devuelve primero la dirección IPv4 93.184.216.34, que es con la que intenta comunicarse. Sin embargo, antes de eso cabría preguntarse porque todo el tráfico DNS está duplicado (ver paquetes #1,3; #5,6; #7,8 y #9,10). Eso se debe a que el equipo está utilizando un servidor DNS local (127.0.0.53), que luego reenvía las peticiones al servidor DNS recursivo del ISP (80.58.61.250), y luego reenvía las respuestas de vuelta. Además, como el servidor DNS se encuentra fuera de la subred local (192.168.1.0/24), es necesario enviar el tráfico DNS al *router* por defecto (192.168.1.1), y para ello hay que realizar una consulta ARP (paquetes #2,4) para obtener su dirección MAC (78:29:ed:dc:62:99). Aunque, gracias al funcionamiento de ARP y de su cache, solo es necesario realizarlo para el primer paquete DNS saliente (paquete #5), y no para el siguiente (paquete #6), ni para las respuestas del servidor DNS remoto (paquetes #7,9). De hecho, como suele haber bastante tráfico con el *router* por defecto, es bastante habitual no capturar resoluciones ARP puesto que su MAC suele estar en la cache (en este caso se ha borrado la caché ARP para mostrar todo el tráfico).

Este ejemplo también muestra la importancia de usar los filtros con cuidado. Por ejemplo, si hubiésemos utilizado un filtro de captura en 'tcpdump' o de visualización en 'Wireshark' que solo nos mostrase el tráfico UDP con el puerto 7000 (udp.port==7000), nos hubiésemos perdido todo este tráfico ARP y DNS previo, y con ello el nombre DNS del servidor destino. Por ello reiterar la recomendación de solo usar los filtros de para localizar el primer paquete de interés, y luego eliminarlo para poder ver tráfico interesante alrededor del mismo.

Volviendo a la captura, a pesar de resolver correctamente la dirección del servidor UDP destino (93.184.216.34/7000) y enviarle mensajes UDP (paquetes #11-13), no obtiene respuesta, por lo que el cliente realiza 3 transmisiones, separadas por unos 3 segundos (ver TIMEOUT en **Listado 4.1**). Podemos estar seguros de que son

retransmisiones del mismo mensaje y no peticiones de otro cliente, porque tienen la misma longitud (65 bytes) y sobre todo porque se envían desde el mismo puerto UDP origen (351030) y por lo tanto desde el mismo *socket*.

Por el contrario, las siguientes peticiones del cliente UDP al servidor UDP local utilizan puertos efímeros diferentes (48289 y 40285), mientras que el servidor mantiene el mismo puerto reservado (7000). Estas transacciones sí tienen éxito, mostrando un datagrama UDP en cada sentido. Solo mencionar que, a pesar de que en la primera de ellas al cliente UDP (paquetes #14-15) también se le pasa el nombre DNS del servidor (localhost), como ese nombre está configurado en el fichero /etc/hosts del equipo local, getaddrinfo() es capaz de resolver la dirección IPv4 asociada (127.0.0.1) sin necesidad de realizar una consulta al servidor DNS local (127.0.0.53) o remoto. Además, merece la pena destacar que aunque el servidor UDP está escuchando en la dirección comodín IPv6 (::) y muestra la dirección IPv4 del cliente embebida en una dirección IPv6 (::ffff:127.0.0.1, ver **Listado 4.16**), el tráfico realmente es IPv4 (paquetes #14-15).

Finalmente, el tráfico del cliente UDP en el equipo 192.168.1.110, también utiliza un puerto efímero (55745) y aunque obtiene respuesta a la primera (paquetes #20-21), requiere una resolución ARP previa (paquetes #18-19) para que pueda obtener la dirección MAC del equipo 192.168.1.33 (00:15:5d:38:01:04).

Para estudiar cómo responde UDP a los errores, también vamos a analizar la captura del escaneo de puertos UDP que se realizó en el **Listado 4.11** utilizando la herramienta 'nmap' desde la misma máquina. Tal como puede observarse en la **Figura 4.18**, 'nmap' va enviando (de manera desordenada para dificultar su detección), un datagrama UPD sin datos a cada puerto que se solicita escanear. Cuando ningún proceso está a la escucha en ese puerto el núcleo genera un mensaje ICMP de Puerto no Alcanzado, incluyendo las cabeceras IPv4 y UDP de forma que 'nmap' sabe qué puerto está cerrado. Sin embargo, en el puerto UDP/7000 donde está escuchando el servidor UDP de ejemplo, se puede ver que el mensaje UDP vacío (paquete #21621) no genera un mensaje ICMP de error, e incluso hay un mensaje UDP de respuesta (#21622), aunque este no suele ser el caso puesto que los servidores UDP reales no suelen responder a mensajes vacíos.

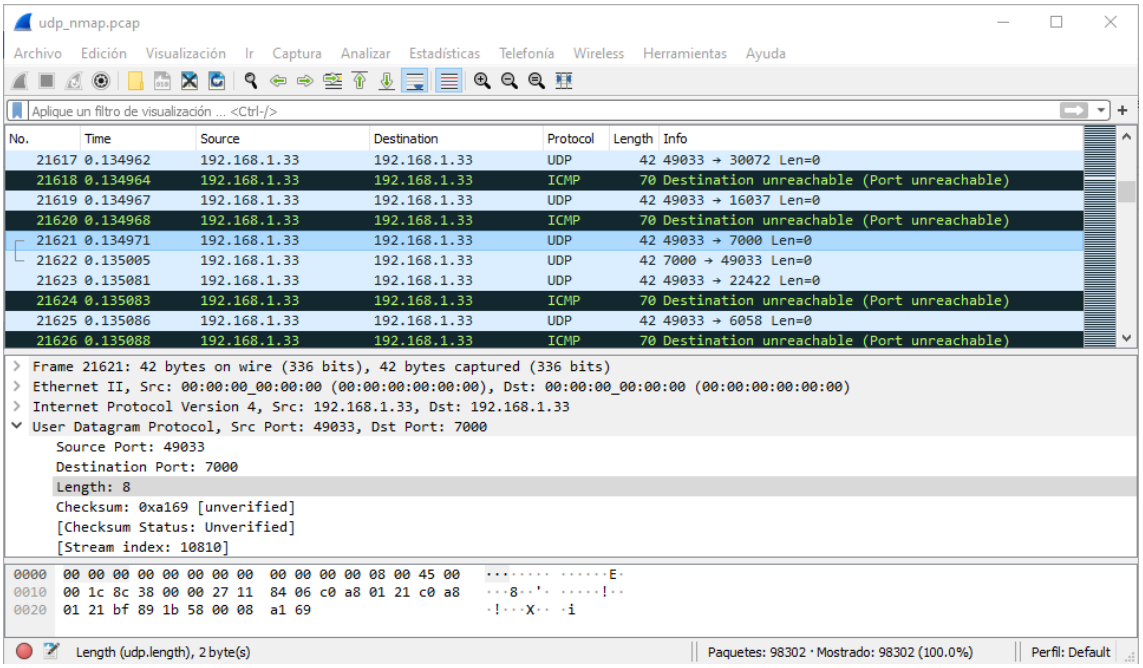



**Figura 4.18** – Captura de tráfico un escaneo de puertos UDP con 'nmap' (udp_nmap.pcap)

# 5. Análisis de aplicaciones basadas en el protocolo TCP

En este capítulo estudiaremos cómo analizar aplicaciones que utilizan el protocolo TCP (*Transmission Control Protocol*). Así que se recomienda repasar el apartado sobre TCP en el **Capítulo 2** de Repaso de la Torre de Protocolos TCP/IP.

## 5.1. Análisis de una aplicación cliente TCP

En este capítulo vamos a analizar una aplicación cliente-servidor TCP de ejemplo. Seguiremos la misma estructura que el capítulo de UDP, asumiendo primero que tenemos acceso al código fuente de la aplicación. Luego, solo al ejecutable de la misma, y finalmente solo el tráfico de red que genera la misma. De esta forma se podrán ir asociando los diferentes eventos que ocurren en las diferentes trazas con el comportamiento de la aplicación. El código fuente completo de todos los ejemplos se distribuye junto a esta memoria, por lo que se recomienda compilarlo y ejecutarlo para analizar el resultado de las diferentes herramientas.

La aplicación cliente-servidor TCP que analizaremos es un servidor de ficheros muy simple. El cliente envía el nombre del fichero que quiere descargar al servidor TCP y almacena todos los datos recibidos en un fichero con el mismo nombre (que no debe existir para evitar problemas cuando el cliente y servidor se prueban en el mismo directorio). Como TCP no permite delimitar mensajes, se utiliza el cierre de cada sentido de la conexión para indicar el fin del nombre del fichero y la descarga del fichero (de una manera similar a la utilizada por HTTP/1.0).

Empecemos analizando el programa cliente. El **Listado 5.1** muestra el código fuente en C del programa, mientras que los **Listados 5.2, 5.3 y 5.4** muestran sendas ejecuciones de dicho programa.

**Listado 5.1** – Código fuente editado del cliente TCP de ejemplo (tcp_client.c)

```
#define BUFFER_SIZE 1024
#define TIMEOUT 10000 /* ms = 3 sec */
#define MAX_RETRIES 2

int main ( int argc, char* argv[] )
{
  if (argc != 4) {
    const char * myself = basename(argv[0]);
    fprintf(stderr, "Usage: %s <addr> <port> <file>\n", myself);
    fprintf(stderr, "  <server>: Server domain name or IP address\n");
    fprintf(stderr, "    <port>: Port number of the TCP server\n");
    fprintf(stderr, "<filename>: Filename to request\n");
    exit(EXIT_FAILURE);
```

```
}
const char*   server_arg = argv[1];
const char*     port_arg = argv[2];
const char* filename_arg = argv[3];

/* Obtain server address(es) matching host/port */
struct addrinfo hints;
memset(&hints, 0x00, sizeof(hints));
hints.ai_family = AF_UNSPEC;     /* Allow IPv4 or IPv6 addresses */
hints.ai_socktype = SOCK_STREAM;  /* Datagram socket */
hints.ai_flags = AI_ADDRCONFIG;
hints.ai_protocol = IPPROTO_TCP; /* TCP protocol */
struct addrinfo *result;

int err = getaddrinfo(server_arg, port_arg, &hints, &result);
if (err != 0) {
  fprintf(stderr, "getaddrinfo() error: %s\n", gai_strerror(err));
  exit(EXIT_FAILURE);
}

/* getaddrinfo() returns a list of address structures.
 * Try each address until we successfullty connect().
 * If socket() of connect() fails, we close the socket
 * and try the next address. */
struct addrinfo *ptr;
int sock_fd;
for (ptr = result; ptr != NULL; ptr = ptr->ai_next) {

  /* Open a TCP socket */
  sock_fd = socket(ptr->ai_family, ptr->ai_socktype,
                   ptr->ai_protocol);
  if (sock_fd == -1) {
    continue; /* Failure! Try next address */
  }

  /* Open a TCP connection to server */
  err = connect(sock_fd, ptr->ai_addr, ptr->ai_addrlen);
  if (err == 0); {
    break; /* Success! No need to keep looking for more addresses */
  }

  close(sock_fd); /* Close socket, because connect() failed */
}

if (ptr == NULL) {
  /* No address succeeded */
  fprintf(stderr, "Cannot connect to TCP Server '%s:%s'\n",
        server_arg, port_arg);
  exit(EXIT_FAILURE);
```

```
}

freeaddrinfo(result); /* No longer needed */

/* Get Server IP address and port */
struct sockaddr_storage server_sa;
socklen_t server_sa_len = sizeof(server_sa);
err = getpeername(sock_fd, (struct sockaddr *) &server_sa,
            &server_sa_len);
if (err == -1) {
  perror("getpeername() error");
  exit(EXIT_FAILURE);
}

char server_addr_str[NI_MAXHOST];
char server_port_str[NI_MAXSERV];
int gni_flags = NI_NUMERICHOST | NI_NUMERICSERV;
err = getnameinfo((struct sockaddr *) &server_sa, server_sa_len,
                  server_addr_str, sizeof(server_addr_str),
                  server_port_str, sizeof(server_port_str),
            gni_flags);
if (err == -1) {
   fprintf(stderr, "getnameinfo() error: %s\n", gai_strerror(err));
   exit(EXIT_FAILURE);
}
printf("Connected to TCP Server %s:%s\n", server_addr_str,
      server_port_str);

/* Create local file to download remote one */
int file_fd = open(filename_arg, O_WRONLY | O_CREAT | O_EXCL,
             S_IRUSR | S_IWUSR);
if (file_fd == -1) {
  if (errno == EEXIST) {
   fprintf(stderr, "File \"%s\" already exists! Please delete it\n",
         filename_arg);
  } else {
   perror("open() error");
  }
  exit(EXIT_FAILURE);
}

/* Send request to TCP server and close transmission channel */
ssize_t bytes_sent = send(sock_fd, filename_arg,
                    strlen(filename_arg), 0);
if (bytes_sent == -1) {
  perror("send() error");
  exit(EXIT_FAILURE);
}
```

```
  err = shutdown(sock_fd, SHUT_WR);
  if (err != 0) {
    perror("shutdown() error");
    exit(EXIT_FAILURE);
  }
  printf("Requesting TCP Server %s:%s to download file \"%s\"\n",
                server_addr_str, server_port_str, filename_arg);

  /* Receive the file from server */
  char buffer[BUFFER_SIZE];
  size_t total_bytes = 0;
  ssize_t bytes_recv;
  do {
    bytes_recv = recv(sock_fd, buffer, sizeof(buffer), 0);
    printf("D: %ld bytes_received\n", bytes_recv);
    if (bytes_recv == -1) {
      perror("recv() error");
      exit(EXIT_FAILURE);
    }
    ssize_t bytes_written = write(file_fd, buffer, bytes_recv);
    if (bytes_written == -1) {
      perror("write() error");
      exit(EXIT_FAILURE);
    }
    total_bytes += bytes_written;

  } while (bytes_recv > 0);

  if (total_bytes == 0) {
    fprintf(stderr, "TCP Server has not sent the requested file\n");
  } else {
    printf("%ld bytes received from TCP Server\n", total_bytes);
  }

  /* Close file, socket and exit */
  err = close(file_fd);
  if (err != 0) {
    perror("close() error");
    exit(EXIT_FAILURE);
  }
  err = close(sock_fd);
  if (err != 0) {
    perror("close() error");
    exit(EXIT_FAILURE);
  }
  return EXIT_SUCCESS;
}
```

**Listado 5.2** – Información de uso del cliente TCP de ejemplo (tcp_client)

```
$ ./tcp_client
Usage: tcp_client <addr> <port> <file>
  <server>: Domain name or IP address of the TCP server
    <port>: Port number of the TCP server
<filename>: Filename to request to the TCP server
```

**Listado 5.3** – Ejecución con éxito del cliente TCP de ejemplo (tcp_client)

```
$ cd test
$ ../tcp_client 127.0.0.1 8000 tcp_client.c
Connected to TCP Server 127.0.0.1/8000
Requesting TCP Server 127.0.0.1/8000 to download file "tcp_client.c"
D: 1024 bytes received
D: 1024 bytes received
D: 1024 bytes received
D: 1024 bytes received
D: 882 bytes received
D: 0 bytes received
4978 bytes received from TCP Server
```

**Listado 5.4** – Ejecución fallida del cliente TCP de ejemplo (tcp_client)

```
$ cd test
$ ../tcp_client 127.0.0.1 8000 file_doesnt_exists.txt
Connected to TCP Server 127.0.0.1/8000
Requesting TCP Server 127.0.0.1/8000 to download file "file_doesnt_exists.txt"
D: bytes_recv = 0
TCP Server has not sent the requested file
```

### 5.1.1. Principales llamadas al sistema de un cliente TCP

A pesar de su simplicidad (especialmente en lo referido a la delimitación de los mensajes de nivel de aplicación) el cliente de ejemplo tiene la estructura que debería cualquier cliente TCP más sofisticado. Analizaremos ahora las llamadas al sistema y funciones de red POSIX que utiliza, aunque sin repetir las ya vistas en el cliente-servidor UDP del capítulo anterior:

- **int socket(int domain, int type, int protocol)** – El cliente TCP crea un *socket* de la familia/dominio AF_UNSPEC para soportar tanto direcciones IPv4 como IPv6, y de tipo **SOCK_STREAM**, puesto que TCP (**IPPROTO_TCP**) es un protocolos fiable. Al ser una aplicación cliente, que no requiere un número de puerto específico, no es necesario realizar un bind(), por lo que se empleará un puerto efímero (i.e. en el rango 49.152-65.535). Como en el cliente UDP se ha analizado en

detalle la posibilidad de utilizar tanto direccione IPv4 como IPv6 gracias a getaddrinfo(), en esta sección no se volverá a incidir en ello.

- **int connect(int sockfd, const struct sockaddr *addr, socklen_t addrlen)** – Esta llamada tiene un comportamiento muy diferente cuando se realiza sobre un socket UDP (que simplemente indica al núcleo que solo se desea comunicarse con un destino), que cuando se realiza sobre un socket TCP. En particular, esta llamada establece una conexión TCP con el destino, esto es, realiza la negociación *3-way handshake* de TCP con el destino. Por lo que no se puede enviar o recibir información hasta que esta llamada se haya completado correctamente. En el código que nos ocupa, las llamadas socket() y connect() se encuentran dentro de un bucle que intenta conectarse a todas las direcciones devueltas por getaddrinfo(), de forma que si alguna se encuentra inaccesible, intenta conectarse a la siguiente.

- **ssize_t send(int sockfd, const void *buf, size_t len, int flags)** - Esta función tiene el mismo comportamiento que en un socket UDP conectado, permite enviar datos al otro extremo de la conexión TCP, aunque en este caso no es necesario que la aplicación se preocupe de la fiabilidad de la transmisión, porque TCP se encargará de entregar de manera fiable los datos. Por esa razón no es necesario que la llamada a send() esté dentro de un bucle que compruebe si hay respuesta del servidor, ni de implementar un temporizador con poll(), como en el cliente UDP. Es más, si no se especifican *flags* adicionales, esta llamada es exactamente igual a llamar a write() sobre el *socket*, como si fuese una simple escritura en un fichero local.

- **int shutdown(int** ***sockfd*****, int** ***how*****)** – Esta función permite indicar que ya no se desea enviar datos (SHUT_WR), recibirlos (SHUT_RD) o ambas cosas (SHUT_RDWR) a través de una conexión TCP establecida. En este caso se utiliza para cerrar el envío de más información al servidor una vez enviado el nombre del fichero a descargar, y de esta forma marcar el fin del nombre del fichero. Puesto que TCP puede dividir cualquier mensaje en múltiples segmentos, y por tanto (a no ser que se defina un mecanismo de delimitación de mensajes a nivel de aplicación) no se puede saber exactamente cuándo ha terminado de transmitirse. Esta llamada enviará un segmento TCP con el *flag* FIN activo al servidor, pero la conexión seguirá activa en el otro sentido.

- **ssize_t recv(int sockfd, void *buf, size_t len, int flags)** – Una vez enviando el nombre del fichero, el cliente inicia un bucle para recibir el contenido del fichero solicitado, guardándolo en un nuevo fichero. Sabemos cuándo se termina de enviar el fichero porque el servidor cierra su sentido de la conexión, lo que genera que recv() devuelva el valor 0. Si la conexión se cierra sin haber

recibido ningún dato anterior, eso quiere decir que el servidor no nos ha enviado el fichero (por ejemplo, porque el fichero solicitado no existe). En el código fuente se ha dejado un mensaje de depuración para ver cómo se va recibiendo la información en múltiples segmentos.

- **int close(int fd)** – Aunque en este momento los dos sentidos de la conexión TCP ya se han cerrado, debe llamarse a la función close(), para liberar los recursos asociados al *socket*. Como este código trabaja simultáneamente con *sockets* y ficheros, es muy fácil ver los paralelismos entre ambos interfaces.

### 5.1.2. Utilización de 'strace' para analizar un cliente TCP

Tal como hicimos en el capítulo anterior con las aplicaciones cliente y servidor UDP, vamos a utilizar el comando 'strace' para analizar las llamadas al sistema que realiza el ejecutable del cliente TCP de ejemplo. De nuevo, vamos a centrarnos únicamente en las llamadas al sistema relacionadas con las operaciones de red (%net) y las que trabajan con descriptores de fichero (%desc).

**Listado 5.5** – Traza de llamadas a sistema del cliente TCP con 'strace' (simplificado)

```
$ strace --trace=%net,%desc -tt -o tcp_client.log ./tcp_client ::1 8000
  "tcp_client.c"
Connected to TCP Server ::1/8000
Requesting TCP Server ::1/8000 to download file "tcp_client.c"
D: 1024 bytes received
D: 1024 bytes received
D: 1024 bytes received
D: 1024 bytes received
D: 882 bytes received
D: 0 bytes received
4978 bytes received from TCP Server

$ cat tcp_client.log
10:20:55.589748 openat(AT_FDCWD, "/etc/ld.so.cache", O_RDONLY|O_CLOEXEC) = 3
10:20:55.589909 fstat(3, {st_mode=S_IFREG|0644, st_size=69530, ...}) = 0
10:20:55.589959 mmap(NULL, 69530, PROT_READ, MAP_PRIVATE, 3, 0) =
                0x7fbeb71f2000
10:20:55.590000 close(3)              = 0
10:20:55.590093 openat(AT_FDCWD, "/lib/x86_64-linux-gnu/libc.so.6",
                 O_RDONLY|O_CLOEXEC) = 3
10:20:55.590146 read(3, "\177ELF\2\1\1\3\0\0\0\0\0\0\0\0\3\0>"..., 832) = 832
10:20:55.590187 pread64(3, "\6\0\0\0\4\0\0\0@\0\0\0\0\0\0"..., 784, 64) = 784
...
10:20:55.590300 fstat(3, {st_mode=S_IFREG|0755, st_size=2029224, ...}) = 0
10:20:55.590339 mmap(NULL, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|
                MAP_ANONYMOUS, -1, 0) = 0x7fbeb71f0000
10:20:55.590387 pread64(3, "\6\0\0\0\4\0\0\0@\0\0\0\0\0\0"..., 784, 64) = 784
...
10:20:55.590500 mmap(NULL, 2036952, PROT_READ, MAP_PRIVATE|MAP_DENYWRITE, 3,
```

```
                0) = 0x7fbeb6ffe000
...
10:20:55.590756 close(3)            = 0

10:20:55.591122 socket(AF_NETLINK, SOCK_RAW|SOCK_CLOEXEC, NETLINK_ROUTE) = 3
10:20:55.591181 bind(3, {sa_family=AF_NETLINK, nl_pid=0, nl_groups=00000000},
                12) = 0
10:20:55.591228 getsockname(3, {sa_family=AF_NETLINK, nl_pid=43276,
                    nl_groups=00000000}, [12]) = 0
10:20:55.591282 sendto(3, {{len=20, type=RTM_GETADDR, flags=NLM_F_REQUEST|
                NLM_F_DUMP, seq=1609286149, pid=0},
                {ifa_family=AF_UNSPEC, ...}}, 20, 0,
                {sa_family=AF_NETLINK, nl_pid=0, nl_groups=00000000},
                12) = 20
10:20:55.591475 recvmsg(3, {msg_name={sa_family=AF_NETLINK, nl_pid=0,
                 nl_groups=00000000}, msg_namelen=12, msg_iov=[...],
                 msg_iovlen=1, msg_controllen=0, msg_flags=0}, 0) = 164
...
10:20:55.591930 socket(AF_UNIX, SOCK_STREAM|SOCK_CLOEXEC|SOCK_NONBLOCK, 0) = 4
10:20:55.591971 connect(4, {sa_family=AF_UNIX, sun_path="/var/run/nscd/socket"
                }, 110) = -1 ENOENT (No existe el archivo o el
            directorio)
10:20:55.592260 close(4)            = 0
10:20:55.592306 close(3)            = 0

10:20:55.592354 socket(AF_INET6, SOCK_STREAM, IPPROTO_TCP) = 3
10:20:55.592405 connect(3, {sa_family=AF_INET6, sin6_port=htons(8000),
                sin6_flowinfo=htonl(0), inet_pton(AF_INET6, "::1",
                &sin6_addr), sin6_scope_id=0}, 28) = 0
10:20:55.592551 getpeername(3, {sa_family=AF_INET6, sin6_port=htons(8000),
                    sin6_flowinfo=htonl(0), inet_pton(AF_INET6, "::1",
                    &sin6_addr), sin6_scope_id=0}, [128->28]) = 0
10:20:55.592626 fstat(1, {st_mode=S_IFCHR|0620, st_rdev=makedev(0x88, 0x3),
                ...}) = 0
10:20:55.592671 write(1, "Connected to TCP Server ::1/8000"..., 33) = 33
10:20:55.592864 openat(AT_FDCWD, "tcp_client.c", O_WRONLY|O_CREAT|O_EXCL,
                0600) = 4
10:20:55.593131 sendto(3, "tcp_client.c", 12, 0, NULL, 0) = 12
10:20:55.593648 shutdown(3, SHUT_WR)    = 0
10:20:55.593448 write(1, "Requesting TCP Server ::1/8000 t"..., 63) = 63
10:20:55.593747 recvfrom(3, "#include <libgen.h>\n#include <st"..., 1024, 0,
                NULL, NULL) = 1024
10:20:55.593807 write(1, "D: 1024 bytes received\n", 23) = 23
10:20:55.594069 write(4, "#include <libgen.h>\n#include <st"..., 1024) = 1024
10:20:55.594217 recvfrom(3, ".ai_family = AF_UNSPEC;     /* A"..., 1024, 0,
                NULL, NULL) = 1024
10:20:55.594283 write(1, "D: 1024 bytes received\n", 23) = 23
10:20:55.594440 write(4, ".ai_family = AF_UNSPEC;     /* A"..., 1024) = 1024
10:20:55.594489 recvfrom(3, "o need to keep looking for more "..., 1024, 0,
                NULL, NULL) = 1024
10:20:55.594569 write(1, "D: 1024 bytes received\n", 23) = 23
10:20:55.594717 write(4, "o need to keep looking for more "..., 1024) = 1024
10:20:55.594766 recvfrom(3, "EXIT_FAILURE);\n  }\n\n  printf(\"Co"..., 1024,
                0, NULL, NULL) = 1024
```

```
10:20:55.594831 write(1, "D: 1024 bytes received\n", 23) = 23
10:20:55.594999 write(4, "EXIT_FAILURE);\n  }\n\n  printf(\""..., 1024) = 1024
10:20:55.595119 recvfrom(3, "recv;\n  do {\n    bytes_recv = re"..., 1024, 0,
                NULL, NULL) = 882
10:20:55.595179 write(1, "D: 882 bytes received\n", 22) = 22
10:20:55.595355 write(4, "recv;\n  do {\n    bytes_recv = re"..., 880) = 880
10:20:55.595414 recvfrom(3, "", 1024, 0, NULL, NULL) = 0
10:20:55.595475 write(1, "D: 0 bytes received\n", 20) = 20
10:20:55.595765 write(4, "", 0)        = 0
10:20:55.595873 write(1, "4978 bytes received from TCP Ser"..., 36) = 36
10:20:55.596150 close(4)              = 0
10:20:55.596195 close(3)              = 0
10:20:55.596380 +++ exited with 0 +++
```

Como el **Listado 5.5** muestra todas las llamadas al sistema que realiza el ejecutable, al principio se muestran las llamadas necesarias para la carga de librerías dinámicas necesarias para su ejecución. Luego (a partir de 10:20:55.591122) aparecen las llamadas al sistema relacionadas con la función getaddrinfo(), que al pasar el *flag* AI_ADDRCONFIG, debe consultar al núcleo, mediante un socket AF_NETLINK, cuáles son las direcciones IP configuradas en los interfaces del sistema, para así solo devolver direcciones compatibles con las mismas. Aunque como se pasa una dirección IPv6 (::1), en lugar de un nombre de DNS, no es necesario leer más ficheros (como el /etc/hosts que aparecía en el mismo listado del cliente UDP).

Esto hace que el cliente TCP cree un socket IPv6 (AF_INET6), de tipo SOCK_STREAM, y con protocolo IPPROTO_TCP. Por lo tanto, analizar las llamadas socket() es la forma más eficiente de ver el tipo de protocolo empleado por una aplicación. Como en esta aplicación se trabaja simultáneamente con este *socket* y con un fichero para almacenar los datos descargados, es importante recordar el descriptor de fichero de cada uno. La llamada socket() devuelve el descriptor de fichero 3, mientras que la función openat() (que es la que realmente se ejecuta a raíz del open() para crear el fichero) devuelve el descriptor de fichero 4 al crear el fichero, mientras que la salida estándar utiliza el descriptor de fichero 1.

Analizando las llamadas al sistema que trabajan sobre el descriptor de fichero 3 (en negrita), se pueden observar de nuevo la mayoría de las funciones de red que aparecen en el código fuente del **Listado 5.1**, aunque de nuevo con ciertas variantes. Por ejemplo, aunque el código fuente utiliza las llamadas send() y recv(), las llamadas al sistema que realmente se ejecutan son sendto() y recvfrom(), aunque sin especificar las direcciones remotas. Además, si el programa no tuviese mensajes de depuración, el valor de bytes leídos devuelto por la función recvfrom() permite identificar la longitud de los segmentos enviados por el servidor: 1.024 + 1.024 + 1.024 + 1.024 + 882 = 4.978 bytes, que es lo que ocupa el fichero descargado, más un valor 0 final que indica que el servidor ha cerrado la conexión.

## 5.2. Análisis de una aplicación servidor TCP

Analicemos ahora una aplicación TCP de tipo servidor, que recibe peticiones de diferentes clientes y les envía una respuesta. En el caso que nos ocupa es un servidor de ficheros muy sencillo, que recibe el nombre de un fichero, y si lo tiene disponible en el directorio desde el que ejecuta, se lo envía al cliente. En caso contrario, cierra la conexión TCP. Obviamente, los servidores TCP reales son bastante más complejos, pero este ejemplo sirve para ilustrar la estructura que debería tener cualquier servidor TCP.

**Listado 5.6** – Código fuente del servidor TCP de ejemplo (tcp_server.c)

```
#define BUFFER_SIZE 4096

#define CLIENT_SUCCESS 0
#define CLIENT_ERROR  -4
#define SERVER_ERROR  -5

int process_client ( int fd, struct sockaddr * sa, ssize_t sa_len );


int main ( int argc, char* argv[] )
{
 if (argc < 2 || argc > 3) {
  const char * myself = basename(argv[0]);
  fprintf(stderr, "Usage: %s <port> [<addr>]\n", myself);
  fprintf(stderr, "   <port>: Port number to bind to\n");
  fprintf(stderr, "   <addr>: IP Address to bind to\n");
  exit(EXIT_FAILURE);
 }

 const char* port_arg = argv[1];
 const char* addr_arg;
 if (argc == 3) {
  addr_arg = argv[2];
 } else {
  addr_arg = NULL;
 }

 /* Obtain server address(es) matching host/port */
 struct addrinfo *result;
 struct addrinfo hints;
 memset(&hints, 0x00, sizeof(hints));
 hints.ai_family = AF_UNSPEC;     /* Allow IPv4 or IPv6 addresses */
 hints.ai_socktype = SOCK_STREAM;  /* Stream socket */
 hints.ai_flags = AI_PASSIVE | AI_NUMERICHOST | AI_ADDRCONFIG;
                  /* Ask for a configured server address */
```

```
hints.ai_protocol = IPPROTO_TCP; /* TCP protocol */

int err = getaddrinfo(addr_arg, port_arg, &hints, &result);
if (err != 0) {
  fprintf(stderr, "getaddrinfo() error: %s\n", gai_strerror(err));
  exit(EXIT_FAILURE);
}

/* getaddrinfo() returns a list of address structures. Try each
 * address until we successfullty bind(). If socket() of bind()
 * fails, we close the socket and try the next address. */
int sock_fd;

struct addrinfo *ptr;
for (ptr=result; ptr!=NULL; ptr=ptr->ai_next) {

  /* Open a TCP socket */
  sock_fd = socket(ptr->ai_family, ptr->ai_socktype,
                   ptr->ai_protocol);
  if (sock_fd == -1) {
    continue; /* Failure! Try next address */
  }

  /* Bind to TCP port */
  err = bind(sock_fd, ptr->ai_addr, ptr->ai_addrlen);
  if (err == 0) {
    break; /* Success! No need to keep looking for more addresses */
  }

  close(sock_fd); /* Close socket, because bind() failed */
}

if (ptr == NULL) {
  /* No address succeeded */
  fprintf(stderr, "Cannot bind to TCP port '%s'\n", port_arg);
  exit(EXIT_FAILURE);
}

freeaddrinfo(result); /* No longer needed */

/* Set SO_REUSEADDR socket option */
int enable = 1;
err = setsockopt(sock_fd, SOL_SOCKET, SO_REUSEADDR,
                 &enable, sizeof(enable));
if (err != 0) {
  perror("setsockopt() error");
  exit(EXIT_FAILURE);
}
```

```
/* Get Server IP address and port */
struct sockaddr_storage server_sockaddr;
socklen_t server_sockaddr_len = sizeof(server_sockaddr);
err = getsockname(sock_fd, (struct sockaddr *) &server_sockaddr,
            &server_sockaddr_len);
if (err == -1) {
  perror("getsockname() error");
  exit(EXIT_FAILURE);
}

char server_addr_str[NI_MAXHOST];
char server_port_str[NI_MAXSERV];
int gni_flags = NI_NUMERICHOST | NI_NUMERICSERV;
err = getnameinfo((struct sockaddr *) &server_sockaddr,
            server_sockaddr_len,
                        server_addr_str, sizeof(server_addr_str),
                        server_port_str, sizeof(server_port_str),
            gni_flags);
if (err == -1) {
  fprintf(stderr, "getnameinfo() error: %s\n", gai_strerror(err));
  exit(EXIT_FAILURE);
}

/* Start listening for incoming TCP connections */
err = listen(sock_fd, 1);
if (err != 0) {
  perror("listen() error");
  exit(EXIT_FAILURE);
}
printf("TCP Server listening at %s/%s ...\n",
      server_addr_str, server_port_str);
printf("Press Ctrl+C to exit.\n");

for(;;) {
  /* Accept connection from client */
  struct sockaddr_storage client_sa;
  socklen_t client_sa_len = sizeof(client_sa);

  int conn_fd = accept(sock_fd, (struct sockaddr *) &client_sa,
                &client_sa_len);
  if (conn_fd == -1) {
    perror("accept()");
    break;
  }

  /* Process client connection */
  int err = process_client(conn_fd, (struct sockaddr*) &client_sa,
                    client_sa_len);
  if (err == SERVER_ERROR) {
```

```
    fprintf(stderr, "Internal TCP Server error. Exiting!\n");
    break;
  } else if (err == CLIENT_ERROR) {
    printf("Error in client request. Closing connection.\n");
  }

  /* Close client connection and wait for next one */
  err = close(conn_fd);
  if (err != 0) {
    perror("close() error");
    break;
  }
 }

 /* Close server socket and exit */
 err = close(sock_fd);
 if (err != 0) {
  perror("close() error");
 }

 return EXIT_FAILURE;
}


int process_client ( int fd, struct sockaddr * sa, ssize_t sa_len )
{
 /* Get client IP address and port */
 char client_addr_str[NI_MAXHOST];
 char client_port_str[NI_MAXSERV];
 int gni_flags = NI_NUMERICHOST | NI_NUMERICSERV;
 int err = getnameinfo(sa, sa_len, client_addr_str,
                sizeof(client_addr_str),
                                client_port_str, sizeof(client_port_str),
                gni_flags);
 if (err == -1) {
  fprintf(stderr, "getnameinfo() error: %s\n", gai_strerror(err));
  return -1;
 }
 printf("New connection from TCP Client %s/%s\n",
       client_addr_str, client_port_str);

 /* Wait until we receive some data from the client.
  * Since TCP may split the response in different segments, concat
  * them until the client closes its end of the connection */
 char buffer[BUFFER_SIZE];
 size_t buffer_left = sizeof(buffer) - 1;
 size_t i = 0;
 ssize_t bytes_recv;
 do {
```

```
  bytes_recv = recv(fd, &buffer[i], buffer_left, 0);
  printf("D: %ld bytes recv\n", bytes_recv);
  if (bytes_recv == -1) {
    perror("recv() error");
    return SERVER_ERROR;
  }

  /* Append received data to buffer (if any) */
  buffer_left -= bytes_recv;
  if (buffer_left <= 0) {
    return SERVER_ERROR; /* Filename too long */
  }
  i += bytes_recv;

} while (bytes_recv > 0);

/* Access only files from current directory */
if (i == 0) {
  return CLIENT_ERROR; /* Empty filename */
} else {
  buffer[i] = '\0';
}
char* filename = basename(buffer);

printf("TCP Client %s/%s has requested file \"%s\"\n",
              client_addr_str, client_port_str, filename);

/* Open requested file */
int file_fd = open(filename, O_RDONLY);
if (file_fd == -1) {
  fprintf(stderr, "Cannot open requested file: \"%s\"\n", filename);
  return CLIENT_ERROR;
}

/* Read file and send it to client until EOF */
ssize_t total_bytes_sent = 0;
ssize_t bytes_read;
do {
  bytes_read = read(file_fd, buffer, sizeof(buffer));
  if (bytes_read == -1) {
    perror("read() error");
    return SERVER_ERROR;

  } else  if (bytes_read > 0) {
    ssize_t bytes_sent = send(fd, buffer, bytes_read, 0);
    printf("D: %ld bytes sent\n", bytes_sent); fflush(stdout);
    if (bytes_sent == -1) {
            perror("send() error");
            return SERVER_ERROR;
```

```
    }

    total_bytes_sent += bytes_sent;
  }
 } while (bytes_read > 0);

 /* File has been successfully sent, print summary and return with
  * success */
 printf("%ld bytes sent to TCP Client %s/%s\n",
            total_bytes_sent, client_addr_str, client_port_str);

 return CLIENT_SUCCESS;
}
```

**Listado 5.7** – Información de uso del servidor TCP de ejemplo (tcp_server)

```
$ ./tcp_server
Usage: tcp_server <port> [<addr>]
  <port>: Port number to bind to
  <addr>: IP Address to bind to
```

**Listado 5.8** – Ejecución del servidor TCP de ejemplo (tcp_server)

```
$ ./tcp_server 8000 ::
TCP Server listening at ::/8000 ...
Press Ctrl+C to exit.
New connection from TCP Client ::1/53062
D: 12 bytes recv
D: 0 bytes recv
TCP Client ::1/53062 has requested file "tcp_server.c"
D: 4096 bytes sent
D: 3112 bytes sent
7208 bytes sent to TCP Client ::1/53062
New connection from TCP Client ::ffff:192.168.1.110/55612
D: 12 bytes recv
D: 0 bytes recv
TCP Client ::ffff:192.168.1.110/55612 has requested file "tcp_client.c"
D: 4096 bytes sent
D: 882 bytes sent
4978 bytes sent to TCP Client ::ffff:192.168.1.110/55612
^C
```

El código del servidor TCP (**Listado 5.6**) es el más complejo que hemos visto hasta ahora y requiere cierta explicación. En particular, además de las nuevas llamadas al sistema (que veremos en el siguiente apartado) y el bucle infinito para recibir peticiones de los clientes (que ya vimos en el servidor UDP), la principal novedad es que las peticiones de los clientes se procesan en una función independiente denominada process_client(). Además de la necesidad obvia de estructurar el código

mejor, aislar todo el código que procesa las peticiones de los clientes en una función es muy habitual en los servidores TCP reales, puesto que permite procesar varias peticiones en procesos o hilos de ejecución en paralelo (aunque este código no lo permite para mantener su simplicidad). Esto es especialmente necesario en los servidores TCP porque las peticiones de los clientes suelen ser más complejas que las peticiones UDP y/o requieren la transferencia de cantidades de información significativas, por lo que utilizar un único hilo/proceso para procesar todas las peticiones de los clientes en orden daría lugar a tiempos de espera elevados.

Otra diferencia fundamental con UDP es que TCP puede entregar los datos agrupados de una forma diferente a la que se enviaron, por lo que es necesario definir algún mecanismo a nivel de aplicación para delimitar los mensajes. De hecho, incluso utilizando el fin de cada sentido de la conexión para marcar el final del nombre del fichero solicitado y del fichero descargado, se requiere código adicional para ir concatenando los datos recibidos en un *buffer* hasta completar el mensaje. La necesidad de este código se puede entender fácilmente comparando las ejecuciones del cliente TCP (**Listado 5.3**) y del servidor TCP (**Listado 5.8**). Como puede observarse, el servidor TCP envía los datos en grupos de hasta 4096 bytes, mientras que el cliente TCP los recibe en grupos de 1024 bytes. Esta diferencia se debe a que el cliente TCP utiliza en su llamada recv() un buffer de 1024 bytes, mientras que el buffer que utiliza el servidor para leer bytes del fichero y enviarlos con un send() tiene un tamaño de 4096 bytes. De hecho, esa información se puede enviar en segmentos de otro tamaño, y aunque igualásemos el tamaño de ambos buffers, TCP podría decidir entregar los datos agrupados de otra forma (siempre que no excedan la ventana de recepción), por ejemplo, debido al tamaño máximo de los segmentos (*Maximum Segment Size* - MSS), el control de flujo o congestión, la retransmisión de segmentos, etc. En particular, para mejorar el tiempo de respuesta de las sesiones interactivas de Telnet o SSH se suele desactivar el algoritmo de Nagle (que agrupa los datos lo más posible antes de enviarlos) para enviar los caracteres uno a uno según se van escribiendo. Por lo tanto, siempre que se trabaje con *sockets* TCP es necesario que las aplicaciones definan un mecanismo de delimitación de mensajes y se encarguen de concatenar los datos recibidos hasta completar un mensaje.

### 5.2.1. Principales llamadas al sistema de un servidor TCP

A continuación, se describen las operaciones más importantes del servidor TCP. Algunas de ellas (como getaddrinfo(), getnameinfo(), freeaddrinfo(), getsockname(), send() o recv()) ya se describieron en el apartado del cliente TCP o en el capítulo sobre UDP, así que no se repetirán aquí.

- **int socket(int domain, int type, int protocol)** – Otro aspecto diferencial de los servidores TCP es que trabajan con varios descriptores de fichero a la vez. La

función socket() se utiliza para crear el *socket* principal del servidor, i.e. el puerto al que se conectan los clientes, sin embargo, la función accept() genera un nuevo descriptor de fichero independiente por cada conexión de un cliente.

- **int bind(int sockfd, const struct sockaddr *addr, socklen_t addrlen)** – La función bind() se utiliza sobre el *socket* principal del servidor para indicar cual es el puerto y opcionalmente la dirección IP desde la que se desea recibir las conexiones de los clientes. El servidor TCP del **Listado 2.10** utiliza el puerto 8000 y la dirección comodín (*wildcard*, en inglés) IPv6 ::, lo que le permite recibir peticiones de clientes tanto IPv6 (::1) como IPv4 (192.168.1.110).

- **int setsockopt(int socket, int level, int option_name, const void *option_value, socklen_t option_len)** – Además de los tres parámetros utilizados para definir el tipo de *socket* a utilizar, los *sockets* pueden configurarse con opciones adicionales, aunque estas dependen del tipo de *socket* y protocolo empleado. Para ello se utilizan las llamadas setsockopt() y getsockopt(), que permiten especificar y consultar, respectivamente, las opciones de un socket. En este caso se establece una opción denominada SO_REUSEADDR, que permite que el servidor reutilice una dirección de transporte que esté bloqueada por una ejecución anterior de otro servidor. Esta opción es bastante habitual en los sockets servidor TCP porque, por defecto las conexiones TCP se mantienen durante un tiempo en memoria (estado TIME_WAIT), incluso después de haberse cerrado, para evitar que paquetes retrasados de las mismas se traten como paquetes de conexiones posteriores. Por lo tanto, por defecto no se puede reutilizar una dirección de transporte hasta que todas las conexiones TCP pendientes hayan caducado (lo que puede tardar varios segundos).

- **int listen(int sockfd, int backlog)** – Esta función, junto con accept(), son llamadas específicas de los servidores TCP. En particular, listen() indica al núcleo que permita recibir TCP conexiones entrantes en ese *socket* (i.e. complete el *3-way handshake* con los clientes), hasta un máximo de *backlog* conexiones en paralelo. Sin embargo, esta función solo pone el *socket* TCP en modo escucha y vuelve inmediatamente, la espera por las conexiones propiamente dichas las realiza la función accept().

- **int accept(int sockfd, struct sockaddr *addr, socklen_t *addrlen)** – Como ya se ha comentado anteriormente, esta función permite esperar una conexión TCP entrante, y cuando esta llega, devuelve la dirección del cliente y sobre todo un descriptor de fichero (que a todos los efectos se comporta como un *socket*) que permite intercambiar información con el otro extremo de la

comunicación. En el código del servidor TCP de ejemplo, la llamada accept() está dentro de un bucle infinito y cuando se recibe una conexión, su socket y la dirección del cliente se pasa a la función process_client() para que se encargue de ella.

- **int close(int fd)** – Además de los close() para cerrar los ficheros solicitados por los clientes, a nivel de red se utilizan dos llamadas close(): la que cierra el *socket* asociado a cada conexión de cliente una vez que se procesa su petición (en este caso no se utiliza la llamada shutdown() como en el cliente, para cerrar la conexión por completo, incluso si el cliente no ha cerrado la suya), y para cerrar el *socket* principal del servidor, aunque esto solo ocurre si se produce un error interno del servidor.

### 5.2.2. Utilización de 'strace' para analizar un servidor TCP

Una vez entendido el código fuente del servidor TDP de ejemplo, vamos a estudiar su traza de llamadas al sistema, obtenidas mediante el comando 'strace', que ya se describió en el mismo apartado del cliente TCP.

**Listado 5.9** – Traza de llamadas a sistema del servidor TCP con 'strace' (simplificado)

```
$ strace --trace=%net,%desc -tt -o tcp_server.log ./tcp_server 8000 ::
TCP Server listening at ::/8000 ...
Press Ctrl+C to exit.
New connection from TCP Client ::ffff:192.168.1.110/53267
D: 12 bytes received
D: 0 bytes received
TCP Client ::ffff:192.168.1.110/53267 has requested file "tcp_client.c"
D: 4096 bytes sent
D: 882 bytes sent
4978 bytes sent to TCP Client ::ffff:192.168.1.110/53267
^C

$ cat tcp_server.log
10:35:47.608765 openat(AT_FDCWD, "/etc/ld.so.cache", O_RDONLY|O_CLOEXEC) = 3
10:35:47.608911 fstat(3, {st_mode=S_IFREG|0644, st_size=69530, ...}) = 0
10:35:47.609000 mmap(NULL, 69530, PROT_READ, MAP_PRIVATE, 3, 0) =
            0x7f3db5c5c000
10:35:47.609050 close(3)              = 0
10:35:47.609093 openat(AT_FDCWD, "/lib/x86_64-linux-gnu/libc.so.6", O_RDONLY|
                  O_CLOEXEC) = 3
10:35:47.609141 read(3, "\177ELF\2\1\1\3\0\0\0\0\0\0\0\0\3\0>"..., 832) = 832
10:35:47.609181 pread64(3, "\6\0\0\0\4\0\0\0@\0\0\0\0\0\0"..., 784, 64) = 784
...
10:35:47.609401 fstat(3, {st_mode=S_IFREG|0755, st_size=2029224, ...}) = 0
10:35:47.609441 mmap(NULL, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|
                MAP_ANONYMOUS, -1, 0) = 0x7f3db5c5a000
10:35:47.609489 pread64(3, "\6\0\0\0\4\0\0\0@\0\0\0\0\0\0"..., 784, 64) = 784
```

```
...
10:35:47.609602 mmap(NULL, 2036952, PROT_READ, MAP_PRIVATE|MAP_DENYWRITE, 3,
                0) = 0x7f3db5a68000
10:35:47.609860 close(3)              = 0


10:35:47.610212 socket(AF_NETLINK, SOCK_RAW|SOCK_CLOEXEC, NETLINK_ROUTE) = 3
10:35:47.610268 bind(3, {sa_family=AF_NETLINK, nl_pid=0, nl_groups=00000000},
                12) = 0
10:35:47.610314 getsockname(3, {sa_family=AF_NETLINK, nl_pid=4555,
                    nl_groups=00000000}, [12]) = 0
10:35:47.610368 sendto(3, {{len=20, type=RTM_GETADDR, flags=NLM_F_REQUEST|
                  NLM_F_DUMP, seq=1609377167, pid=0},
                  {ifa_family=AF_UNSPEC, ...}}, 20, 0,
                  {sa_family=AF_NETLINK, nl_pid=0, nl_groups=00000000},
                  12) = 20
10:35:47.610541 recvmsg(3, {msg_name={sa_family=AF_NETLINK, nl_pid=0,
                   nl_groups=00000000}, msg_namelen=12, msg_iov=[...],
                   msg_iovlen=1, msg_controllen=0, msg_flags=0}, 0) = 164
...
10:35:47.611090 socket(AF_UNIX, SOCK_STREAM|SOCK_CLOEXEC|SOCK_NONBLOCK, 0) = 4
10:35:47.611135 connect(4, {sa_family=AF_UNIX,
                   sun_path="/var/run/nscd/socket"}, 110)
            = -1 ENOENT (No existe el archivo o el directorio)
10:35:47.611388 close(4)              = 0
10:35:47.611429 close(3)              = 0


10:35:47.611476 socket(AF_INET6, SOCK_STREAM, IPPROTO_TCP) = 3
10:35:47.611518 bind(3, {sa_family=AF_INET6, sin6_port=htons(8000),
                sin6_flowinfo=htonl(0), inet_pton(AF_INET6, "::",
                &sin6_addr), sin6_scope_id=0}, 28) = 0
10:35:47.611563 setsockopt(3, SOL_SOCKET, SO_REUSEADDR, [1], 4) = 0
10:35:47.611602 getsockname(3, {sa_family=AF_INET6, sin6_port=htons(8000),
                    sin6_flowinfo=htonl(0), inet_pton(AF_INET6, "::",
                    &sin6_addr), sin6_scope_id=0}, [128->28]) = 0
10:35:47.611657 listen(3, 1)          = 0
10:35:47.611697 fstat(1, {st_mode=S_IFCHR|0620, st_rdev=makedev(0x88, 0),
                ...}) = 0
10:35:47.611739 write(1, "TCP Server listening at ::/8000 "..., 36) = 36
10:35:47.611903 write(1, "Press Ctrl+C to exit.\n", 22) = 22
10:35:47.612097 accept(3, {sa_family=AF_INET6, sin6_port=htons(53267),
                  sin6_flowinfo=htonl(0), inet_pton(AF_INET6,
                  "::ffff:192.168.1.110", &sin6_addr), sin6_scope_id=0},
                  [128->28]) = 4
10:36:05.316700 write(1, "New connection from TCP Client :"..., 58) = 58
10:36:05.316952 recvfrom(4, "tcp_client.c", 4095, 0, NULL, NULL) = 12
10:36:05.317287 write(1, "D: 12 bytes received\n", 21) = 21
10:36:05.317463 recvfrom(4, "", 4083, 0, NULL, NULL) = 0
10:36:05.317569 write(1, "D: 0 bytes received\n", 20) = 20
10:36:05.317824 write(1, "TCP Client ::ffff:192.168.1.110/"..., 72) = 72
10:36:05.317994 openat(AT_FDCWD, "tcp_client.c", O_RDONLY) = 5
10:36:05.318040 read(5, "#include <libgen.h>\n#include <st"..., 4096) = 4096
10:36:05.318079 sendto(4, "#include <libgen.h>\n#include <st"..., 4096, 0,
```

```
                    NULL, 0) = 4096
10:36:05.318287 write(1, "D: 4096 bytes sent\n", 19) = 19
10:36:05.318899 read(5, "recv;\n  do {\n    bytes_recv = re"..., 4096) = 882
10:36:05.319010 sendto(4, "recv;\n  do {\n    bytes_recv = re"..., 882, 0,
                    NULL, 0) = 882
10:36:05.319223 write(1, "D: 882 bytes sent\n", 18) = 18
10:36:05.319326 read(5, "", 4096)       = 0
10:36:05.319351 write(1, "4978 bytes sent to TCP Client ::"..., 57) = 57
10:36:05.319600 close(4)                = 0
10:36:05.319644 accept(3, 0x7fff03c480a0, [128])
              = ? ERESTARTSYS (To be restarted if SA_RESTART is set)
10:36:12.223150 --- SIGINT {si_signo=SIGINT, si_code=SI_KERNEL} ---
10:36:12.223619 +++ killed by SIGINT +++
```

Centrándonos directamente en las llamadas de red del final del **Listado 5.9**, a pesar de la complejidad adicional del código fuente del servidor TCP, es muy fácil analizar su comportamiento. La única dificultad radica en que estas llamadas al sistema utilizan tres descriptores de fichero simultáneamente (además del descriptor de fichero 1, que es la salida estándar). El *socket* principal del servidor utiliza el descriptor 3, que devuelve la llamada a socket() inicial, y que luego se utiliza únicamente para llamar a bind(), setsockopt(), getsockopt(), listen() y accept(), mientras que todas las llamadas a recvfrom() y sendto() para recibir y enviar información al cliente se realizan únicamente sobre el descriptor de fichero 4, que es el que devuelve la llamada a accept() cuando recibe una conexión nueva, y que se cierra con el close() final, cuando se termina de enviar el fichero solicitado al cliente. El descriptor de fichero 5 lo devuelve la llamada a openat() y se utiliza para leer del fichero local.

Merece la pena destacar de nuevo la diferencia entre las llamadas listen() y accept(). La primera sólo se llama una vez y retorna inmediatamente, mientras que accept() se debe ejecutar cada vez que se quiera recibir una conexión TCP y se bloquea hasta que se reciba una. Esto se puede comprobar en la llamada a accept() que ocurre en el instante 10:35:47.612097, mientras que la siguiente llamada al sistema no ocurre hasta el instante 10:36:05.316700, 18 segundos más tarde. En este código es fácil identificar el tiempo entre llamadas al sistema gracias a los profusos mensajes de estado y depuración, en otras circunstancias se puede utilizar la opción '-T' de 'strace' que indica el tiempo que tarda en ejecutar cada llamada al sistema.

Para saber en qué puerto está escuchando un servidor TCP y las conexiones TCP establecidas por los procesos del sistema, se puede utilizar el comando 'netstat', tal como muestra el **Listado 5.10**, que nos permite identificar que, además del servidor TCP de ejemplo corriendo en el puerto TCP/8000, hay un servidor de SSH (TCP/22) escuchando tanto conexiones IPv4 como IPv6, así como el resolver DNS local, que también admite conexiones TCP aunque solo escucha en la dirección de *loopback*. Además de los servidores, se puede observar una conexión TCP establecido contra el servidor SSH desde la máquina 192.168.1.110.

Para intentar detectar los puertos TCP abiertos (i.e. que han llamado a listen()), se puede utilizar el comando 'nmap', tal como muestra el **Listado 5.11**.

**Listado 5.10** – Información del comando 'netstat' sobre servidores y conexiones TCP

```
$ netstat -tanp
Conexiones activas de Internet (servidores y establecidos)
Proto Recib Enviad Dir. local     Dir. remota        Estado     PID/Program
tcp       0      0 127.0.0.53:53   0.0.0.0:*          ESCUCHAR    -
tcp       0      0 0.0.0.0:22      0.0.0.0:*          ESCUCHAR    -
tcp       0      0 192.168.1.33:22 192.168.1.110:52597 ESTABLECIDO -
tcp6      0      0 :::22           :::*               ESCUCHAR    -
tcp6      0      0 :::8000         :::*               ESCUCHAR   26032/
                                                      tcp_server
```

**Listado 5.11** – Información del comando 'nmap' sobre los puertos TCP abiertos

```
$ sudo nmap -sT -n -p1-49151 192.168.1.33
Starting Nmap 7.80 ( https://nmap.org )
Nmap scan report for 192.168.1.33
Host is up (0.000063s latency).
Not shown: 49149 closed ports
PORT     STATE SERVICE
22/tcp   open  ssh
8000/tcp open  http-alt

Nmap done: 1 IP address (1 host up) scanned in 0.63 seconds
```

## 5.3. Análisis del tráfico de una aplicación cliente-servidor TCP

Una vez analizadas las aplicaciones cliente y servidor TCP por separado, vamos a analizar el comportamiento completo de la aplicación y de su protocolo.

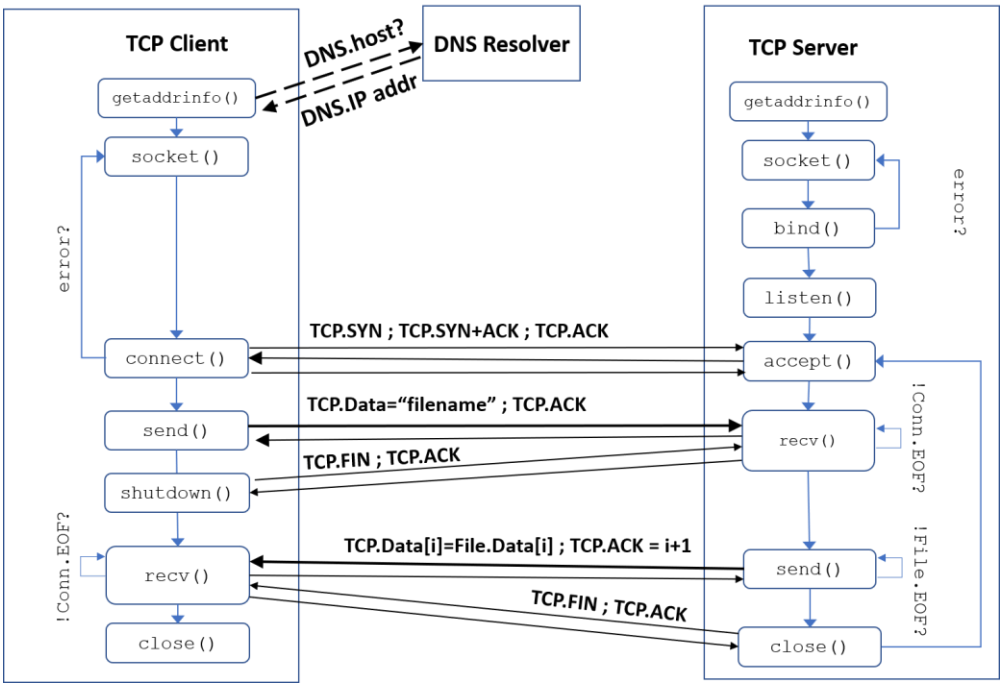



**Figura 5.12** – Modelo de la aplicación cliente-servidor TCP de ejemplo

La **Figura 5.12** muestra un modelo simplificado del flujo de ejecución de las aplicaciones cliente y servidor, así como el tráfico que envía cada una de ellas.

A pesar de la aparente simplicidad de la aplicación de ejemplo (i.e. enviar un nombre de fichero y recibir el contenido del mismo), el comportamiento de la misma en la red es bastante más complejo, especialmente si se compara con la del cliente servidor UDP, que solo generaba dos datagramas UDP en cada sentido. En TCP, sin embargo, antes de enviar cualquier tipo de información es necesario establecer una conexión (i.e. *three-way-handshake*), que en el lado cliente se lanza con connect(), y en el lado servidor se acepta con accept(), después de haber puesto el servidor en modo escucha con listen(). Es importante recordar que en UDP la función connect() no generaba tráfico de red, así que cuando se observe esta llamada al sistema es muy importante saber si se realiza sobre un *socket* TCP o UDP.

El envío y recepción de datos se sigue realizando utilizando send() y receive(), pero en este caso no siempre hay un único segmento TCP por mensaje, sino que TCP puede dividir un mensaje en múltiples segmentos de datos (e.g. el servidor envía mensajes de 4 KiB, pero TCP los entrega en 4 mensajes de 1 KiB, debido al buffer de recepción del cliente), y además la recepción de datos genera paquetes de confirmación

(TCP.ACK), que en este caso no contienen datos. Nótese que este comportamiento se debe a que esta aplicación de ejemplo es fundamentalmente *semi-duplex*, dado que en cada momento solo envía datos el cliente o el servidor. En aplicaciones *full-duplex*, con envío de información en ambos sentidos, no es necesario generar paquetes de confirmación sin datos, porque los segmentos de datos también contienen el número de confirmación del sentido contrario. TCP tiene varios algoritmos para decidir cuándo se debe enviar una confirmación (varios segmentos de datos se pueden confirmar con un único ACK), así que en aplicaciones más complejas no es nada fácil adivinar el tráfico que va a generar exactamente TCP al enviar un mensaje o recibir un segmento. Por otro lado, que TCP se encargue de garantizar la fiabilidad de los datos, ahorra a las aplicaciones TCP más simples de implementar temporizadores (ya no aparece ninguna llamada poll()) y mecanismos de control de errores, tal como era necesario en el cliente UDP.

Por el contrario, la necesidad de implementar mecanismos de delimitación de mensajes a nivel de aplicación, hace que la recepción de datos sea bastante más compleja, siendo habitualmente necesario incluir la llamada a recv() en un bucle que permita recibir múltiples segmentos y concatenarlos hasta completar un mensaje. En este sentido, esta aplicación nos permite observar el cierre parcial de una conexión TCP, puesto que el cliente utiliza la función shutdown() para cerrar su lado de la conexión y así marcar el final del nombre de fichero solicitado. El servidor por su parte, cierra su conexión utilizando el close(), en lugar del shutdown(), pero este comportamiento puede cambiar mucho entre aplicaciones. En particular, en aplicaciones *full-duplex*, donde se envía información en ambos sentidos o en protocolos más avanzados como HTTP/1.1, ambos sentidos de la conexión se mantienen abiertos simultáneamente, y normalmente cuando un extremo cierra la conexión el otro hace lo mismo, dando lugar a un cierre ordenado con cuatro pasos: TCP.FIN A→B, TCP.ACK B→A, TCP.FIN B→A, TCP.ACK A→B.

Pasemos ahora a analizar el tráfico de red generado por esta aplicación. La **Figura 5.15** muestra una captura de tráfico de red realizada desde el equipo 192.168.1.33 (sudo tcpdump -i eth0 -s0 -w tcp.pcap), donde está ejecutando el servidor TCP (ver **Listado 5.14**). Mientras que el cliente TCP ejecuta desde el equipo 192.168.1.110 (ver **Listado 5.13**), que está en la misma subred (192.168.1.0/24).

**Listado 5.13** – Configuración y ejecución del cliente TCP de ejemplo en el equipo 192.168.1.110

```
muruenya@192.168.1.110:~$ route -n
Kernel IP routing table
Destination     Gateway         Genmask         Flags Metric Ref    Use Iface
0.0.0.0         192.168.1.1     255.255.255.255 U     0      0        0 eth2
192.168.1.110   0.0.0.0         255.255.255.255 U     256    0        0 eth2

muruenya@192.168.1.110:~/test/$ ../tcp_client 192.168.1.33 8000 tcp_client.c
```

```
Connected to TCP Server 192.168.1.33/8000
Requesting TCP Server 192.168.1.33/8000 to download file "tcp_client.c"
D: bytes_recv = 1024
D: bytes_recv = 1024
D: bytes_recv = 1024
D: bytes_recv = 1024
D: bytes_recv = 882
D: bytes_recv = 0
4978 bytes received from TCP Server
```

**Listado 5.14** – Ejecución del servidor TCP de ejemplo en el equipo 192.168.1.33

```
muruenya@192.168.1.33:~$ ./tcp_server 8000 ::
TCP Server listening at ::/8000 ...
Press Ctrl+C to exit.
New connection from TCP Client ::ffff:192.168.1.110/64118
D: 12 bytes received
D: 0 bytes received
TCP Client ::ffff:192.168.1.110/64118 has requested file "tcp_client.c"
D: 4096 bytes sent
D: 882 bytes sent
4978 bytes sent to TCP Client ::ffff:192.168.1.110/64118
```

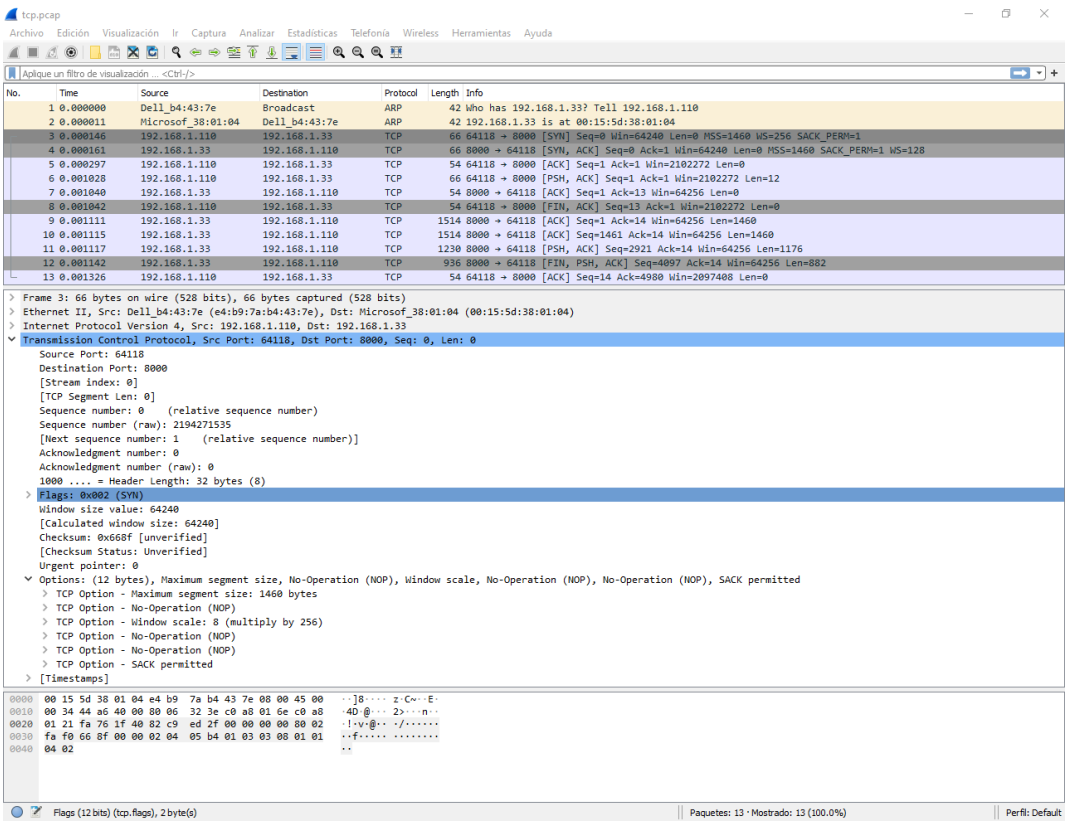


**Figura 5.15** – Captura de tráfico de la aplicación cliente-servidor TCP de ejemplo (tcp.pcap)

Al estar en la misma subred, los dos primeros paquetes de la captura (#1-2) son la petición y respuesta ARP que realiza el cliente (192.168.1.110) para conocer la dirección MAC del servidor (192.168.1.33). No es necesaria la consulta inversa, porque en la petición ARP el cliente incluye su propia dirección MAC, que el servidor introduce en su caché ARP, y por tanto tiene disponible para cuando necesita enviar tráfico de vuelta.

Así que el primer paquete (#3) que realmente ha sido generado por el cliente es el seleccionado en 'Wireshark'. Es el primer paquete del *three-way handshake* generado por la llamada a connect(). Tal como puede observarse sólo tiene activado el *flag* SYN y un número de secuencia, mientras que el campo de ACK está a 0. Un comentario sobre los números de secuencia de TCP que muestra 'Wireshark': aunque realmente las conexiones TCP utilizan números de secuencia que comienzan con un valor aleatorio (e.g. 2194271535) para evitar varios tipos de ataques, 'Wireshark' suele mostrar números de secuencia y ACK relativos (e.g. 0), para que sea más fácil interpretar los segmentos TCP. Además, la cabecera TCP del primer segmento SYN incluye tres opciones:

- *Maximum Segment Size* (MSS) = 1460 bytes. Esta opción indica que el emisor desea recibir segmentos TCP con una longitud máxima de 1460 bytes. Este es un valor bastante habitual para interfaces de red Ethernet, que tienen una MTU (*Maximum Transfer Unit*) de 1500 bytes = 1460 bytes de segmento TCP + 40 bytes de cabecera IPv6.
- *Windows Scale* = 8 (256). Esta opción indica que el tamaño de ventana anunciado en el campo *Window size* (i.e. 64.240) debe multiplicarse por $2^8$=256, lo que significa que el cliente tiene una ventana de recepción de 64.240 * 256 = 16.445.440 bytes = 15,68 MiB. Sin esta opción, el tamaño de la ventana de recepción de TCP estaría limitado a un máximo de $2^{16}$ = 64 KiB, lo que limitaría mucho su rendimiento en redes de alta velocidad y latencia, porque esa es la cantidad máxima de datos que pueden estar en vuelo sin recibir confirmación. Nótese que el valor de la ventana de recepción de TCP es mucho mayor que el tamaño del buffer definido por el cliente (1024 bytes).
- *Selective Acknowledgement (SACK)*. Esta opción indica que el emisor soporta el mecanismo de confirmaciones selectivas, lo que le permite indicar que segmentos ha recibido y cuales no de manera independiente, en lugar de únicamente el último número de ACK, lo que en caso de pérdidas individuales de segmentos podría dar lugar a retrasmisiones innecesarias.
- Las opciones *No-Operation* (NOP) solo se utilizan para alinear el resto de opciones a palabras de 32 bits, puesto que la opción de *Windows Scale* ocupa 3 bytes, y la de SACK ocupa 2 bytes.

El siguiente segmento del *three-way handshake* (#4) es el que envía el servidor y tiene activos los *flags* SYN y ACK, confirmando el número de secuencia del segmento anterior + 1 (por eso aparece el valor relativo Ack=1 en el campo 'Info' del paquete). Aunque no se muestra el paquete completo, del campo 'Info' se puede observar que también incluye las mismas 3 opciones que envía el cliente, por lo que su MSS también es de 1460 bytes, también soporta el mecanismo de SACK, aunque tiene un tamaño de ventana de recepción inferior al cliente: 64.240 * 128 = 8.222.720 bytes = 7,84 MiB.

El *three-way handshake* se completa con el ACK (#5) que vuelve a enviar el cliente, confirmado el número de secuencia enviado por el servidor + 1 y aumentando su propio número de secuencia (los *flags* SYN y FIN aumentan en 1 el número de secuencia). Aunque al mostrar únicamente números de secuencia relativos en el campo 'Info', solo aparecen como Seq=1 Ack=1. Pero si se analiza el fichero de captura (tcp.pcap), se puede observar que los números de secuencia y ACK reales son 2194271536 y 2328673255.

Curiosamente el siguiente segmento de la conexión TCP (#6), ya con datos, también lo envía el cliente, por lo que estos se podrían incluir en el ACK anterior. Sin embargo este es el comportamiento habitual de las implementaciones POSIX, debido a que el establecimiento de conexión y el envío de información están separadas en dos funciones diferentes: connect() y send(), respectivamente. Ese segmento transporta los 12 bytes del nombre de fichero solicitado "tcp_client.c" y tiene el *flag* PSH activo, que indica al destino que debería entregar esos datos lo antes posible a la aplicación. La recepción correcta de esos datos se confirma con un segmento sin datos (#7) con número de ACK=13. Luego el cliente manda otro segmento sin datos (#8) pero con el *flag* FIN activo, que se genera al llamar a shutdown(), e indica al servidor que ya no va a enviar más datos (y que la aplicación TCP de ejemplo utiliza para marcar el final del nombre del fichero solicitado). La confirmación de este segmento de FIN no genera un ACK sin datos, sino que se aprovecha el siguiente segmento de datos que envía el servidor (con ACK=14).

Y es que el servidor manda el fichero solicitado (que ocupa 4.978 bytes) en 4 segmentos consecutivos (#9-12) de 1.460 + 1.460 + 1.176 + 882 bytes. Esto se debe a que la aplicación manda el fichero en grupos de 4.096 bytes, pero el MSS del cliente es solo de 1.460 bytes, por lo que tiene que enviar esos 4.096 bytes en 3 segmentos de 1.460 + 1.460 + 1.176 = 4.096 bytes. El resto de fichero solo ocupa 4.978 – 4.096 = 882 bytes, que ya caben en un único segmento. Esta explicación se ve confirmada por los *flags* PSH que tienen los paquetes #11 y #12, y que coincidirían con las 2 llamadas send() que realiza el servidor. Pero como ya hemos repetido en varias ocasiones, nada impediría que el servidor enviase los mismos datos de otra forma, como por ejemplo en segmentos de 1.460 + 1.460 + 1.460 + 598 bytes.

De hecho, aunque en el cliente la llamada a shutdown() genera un segmento FIN vacío (#8), en este caso la llamada a close() del servidor para marcar el final del fichero simplemente activa el *flag* FIN del último segmento de datos (#12). Que el cliente confirma con el último paquete de la conexión que es un ACK vacío (puesto que el cliente ya no puede enviar datos), y que sirve para confirmar tanto ese FIN, como todos los datos recibidos hasta el momento con el número de secuencia 4980 = 1 (SYN) + 4.978 (Datos) + 1 (FIN).

Obviamente, tener los datos intercambiados en múltiples paquetes complica su análisis. Por eso es tan útil la opción 'Seguir > Flujo TCP (Ctrl + Alt + Shift + T)', que permite observar todos los datos intercambiados, tal como se muestra en la **Figura 5.16**. Los datos de cada sentido de la conexión utilizan un color diferente. Por lo que en rojo se puede ver el nombre del fichero solicitado (tcp_client.c), mientras que en azul se puede ver el contenido completo del fichero descargado.

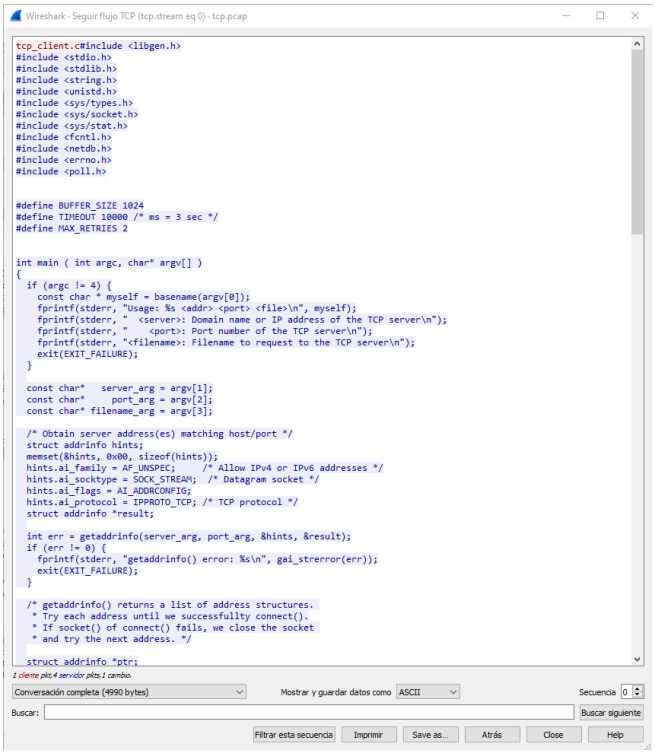



**Figura 5.16** – Información del flujo TCP capturado (tcp.pcap)

**Listado 5.17** – Configuración y ejecución del cliente TCP de ejemplo en el equipo 192.168.1.110

```
muruenya@192.168.1.110:~/test/$ ../tcp_client 192.168.1.33 8888 tcp_server.c
Cannot connect to TCP Server '192.168.1.33/8888'
```

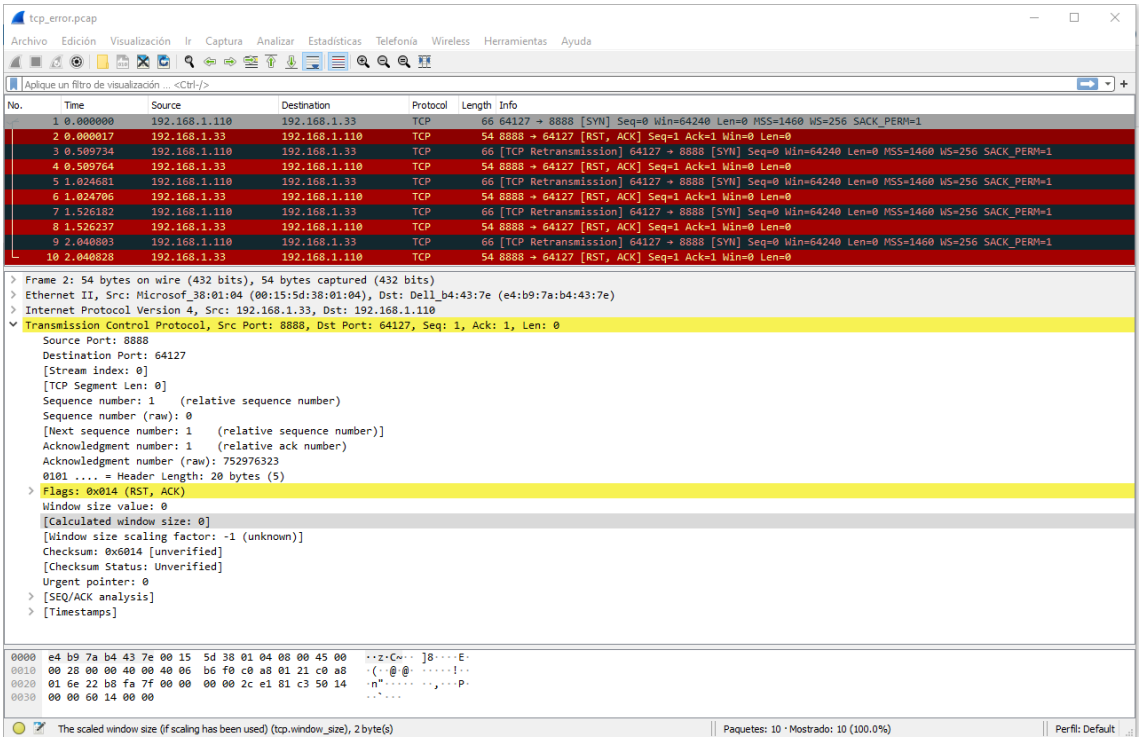



**Figura 5.18** – Captura de una conexión TCP infructuosa (tcp_error.pcap)

Por último, vamos a analizar una pequeña captura de tráfico erróneo, generado porque el cliente (**Listado 5.17**) intenta conectarse al puerto 8888 del servidor, en lugar del puerto 8000 donde realmente está escuchando el servidor (**Listado 5.10**).

La **Figura 5.18** muestra el comportamiento en la red. El cliente intenta establecer la conexión enviando el primer segmento SYN (paquete #1) del *three-way handshake*, que es exactamente igual al mostrado en la **Figura 5.15**, salvo por el puerto destino y el número de secuencia aleatorio. Sin embargo, como respuesta ya no recibe un segmento con los *flags* SYN + ACK, sino que en este caso recibe un segmento (paquete #2) con los *flags* RST y ACK. El *flag* ACK simplemente indica que el segmento contiene un número de ACK valido (que coincide con el número de secuencia enviado por el cliente + 1), pero el *flag* RST indica que la conexión debe cerrarse inmediatamente, o en este caso no llegar a establecerse. El mismo intercambio se repite 4 veces, separadas unos 500 ms, hasta que la llamada connect() del cliente devuelve un error. El escaneo de puertos TCP con el comando 'nmap' mostraría una captura de tráfico similar (aunque con un único intento de conexión por puerto).

# 6. Análisis de aplicaciones basadas en *sockets* "crudos"

Aunque la gran mayoría de las aplicaciones utilizan los protocolos de transporte (e.g. TCP o UDP) que implementa el sistema anfitrión, muchos sistemas operativos permiten a las aplicaciones enviar paquetes a bajo nivel (i.e. saltándose una o más capas de la torre de protocolos), por ejemplo, para implementar en espacio de usuario protocolos que todavía no han sido implementado en el núcleo (e.g. las primeras implementaciones de SCTP). Para ello se emplean los conocidos como *sockets* "crudos" (*raw socket*, en inglés), que permiten enviar paquetes de diferentes protocolos directamente. Por ejemplo, se pueden enviar paquetes IPv4, IPv6, ICMP, ICMPv6, UDP, TCP, e incluso tramas Ethernet directamente, sin pasar por las capas superiores de la torre de protocolo, aunque normalmente sí se utilizan las capas inferiores. Por ejemplo, si se envía un datagrama UDP, éste se encamina utilizando la tabla de rutas IP del sistema.

Las capacidades de los *sockets* "crudos" dependen mucho de cada sistema operativo, pero es bastante habitual que para poder usarlos se necesiten permisos de superusuario.

Para ilustrar las capacidades de los *sockets* "crudos" vamos a analizar una pequeña aplicación que es capaz de generar mensajes ICMP de tipo *Echo*, los mismos que utiliza el comando 'ping' para comprobar la conectividad IP con un equipo destino. En este caso utilizaremos la implementación ICMP del equipo destino como servidor para generar los mensajes *Echo Reply* de vuelta, así que en este caso sólo tendremos una única aplicación cliente.

## 6.1. Análisis de una aplicación ICMP de ejemplo

El **Listado 6.1** muestra el código fuente en C del programa, mientras que los **Listados 6.2** y **6.3** muestran sendas ejecuciones de dicho programa.

**Listado 6.1** – Código fuente editado de la aplicación ICMP de ejemplo (icmp.c)

```
#define ICMP_ECHO_LEN 64
#define BUFFER_LEN 1480
#define TIMEOUT 1000 /* ms = 1 sec */

/* Header of ICMP Echo message [RFC792] */
struct icmp_echo_hdr {
  uint8_t type;
  uint8_t code;
  uint16_t checksum;
  uint16_t identifier;
  uint16_t seq_num;
  unsigned char data[];
};
```

```
uint16_t ipv4_checksum ( const void * data, size_t len );

int main ( int argc, char* argv[] )
{
  if (argc != 2) {
    const char * myself = basename(argv[0]);
    fprintf(stderr, "Usage: %s <dest>\n", myself);
    fprintf(stderr, " <dest>: Destination address\n");
    exit(EXIT_FAILURE);
  }
  const char* dest_arg = argv[1];

  /* Obtain destination address */
  struct addrinfo hints;
  memset(&hints, 0x00, sizeof(hints));
  hints.ai_family = AF_INET;     /* Return IPv4 addresses */
  hints.ai_socktype = SOCK_RAW;  /* Raw socket */
  hints.ai_flags = 0;
  hints.ai_protocol = IPPROTO_ICMP; /* ICMP protocol */
  struct addrinfo *result;

  int err = getaddrinfo(dest_arg, NULL, &hints, &result);
  if (err != 0) {
    fprintf(stderr, "getaddrinfo() error: %s\n", gai_strerror(err));
    exit(EXIT_FAILURE);
  }

  /* Open a RAW socket */
  int sock_fd = socket(result->ai_family, result->ai_socktype,
                result->ai_protocol);
  if (sock_fd == -1) {
    fprintf(stderr,
          "Cannot open RAW Socket. Are you root (uid=0)?\n");
    exit(EXIT_FAILURE);
  }

  /* Set socket option to do not provide IPv4 header. */
  const int off = 0;
  err = setsockopt(sock_fd, IPPROTO_IP, IP_HDRINCL, &off,
              sizeof(off));
  if (err != 0) {
    perror ("setsockopt() error");
    exit (EXIT_FAILURE);
  }

  struct sockaddr_in dest_sa;
  memcpy(&dest_sa, result->ai_addr, result->ai_addrlen);
```

```
freeaddrinfo(result); /* No longer needed */

/* Fill ICMP Echo message */
unsigned char msg[ICMP_ECHO_LEN];
struct icmp_echo_hdr * icmp_echo = (struct icmp_echo_hdr *) &msg;
icmp_echo->type = 0x08;
icmp_echo->code = 0x00;
icmp_echo->checksum = htons(0x0000);
icmp_echo->identifier = htons(0x1337);
icmp_echo->seq_num = htons(0x0001);
int data_len = ICMP_ECHO_LEN - 8;
int i;
for (i=0; i<data_len; i++) {
  icmp_echo->data[i] = i;
}
icmp_echo->checksum = htons(ipv4_checksum(msg, ICMP_ECHO_LEN));

/* Send ICMP Echo to destination address */
ssize_t bytes_sent = sendto(sock_fd, msg, sizeof(msg), 0,
                                        (struct sockaddr *) &dest_sa,
                    sizeof(dest_sa));
if (bytes_sent == -1) {
  perror("sendto() error");
  exit(EXIT_FAILURE);
}

char dest_addr_str[NI_MAXHOST];
int gni_flags = NI_NUMERICHOST;
err = getnameinfo((struct sockaddr *) &dest_sa, sizeof(dest_sa),
                              dest_addr_str, sizeof(dest_addr_str), NULL, 0,
              gni_flags);
if (err == -1) {
   fprintf(stderr, "getnameinfo() error: %s\n", gai_strerror(err));
   exit(EXIT_FAILURE);
}
printf("ICMP Echo (%ld bytes) sent to %s\n",
      bytes_sent, dest_addr_str);

/* Now wait until we have a response.
 * We use poll() to implement a timeout */
nfds_t num_poll_fds = 1;
struct pollfd poll_fds[num_poll_fds];
poll_fds[0].fd = sock_fd;
poll_fds[0].events = POLLIN | POLLERR;

int retval = poll(poll_fds, num_poll_fds, TIMEOUT);
if (retval == -1) {
  perror("poll() error");
  exit(EXIT_FAILURE);
```

```
  } else if (retval == 0) {
    printf("No response received\n");
    exit(EXIT_FAILURE);
  }

  /* Receive the reply.
   * TODO: Check that it comes from the destination
   * and is an Echo Reply */
  char buffer[BUFFER_LEN];
  struct sockaddr_in from_sa;
  socklen_t from_sa_len = sizeof(from_sa);
  ssize_t bytes_recv = recvfrom(sock_fd, buffer, sizeof(buffer), 0,
                                                (struct sockaddr *) &from_sa,
                          &from_sa_len);
  if (bytes_recv == -1) {
    perror("recvfrom() error");
    exit(EXIT_FAILURE);
  }

  char from_addr_str[NI_MAXHOST];
  gni_flags = NI_NUMERICHOST;
  err = getnameinfo((struct sockaddr *) &from_sa, sizeof(from_sa),
                                from_addr_str, sizeof(from_addr_str), NULL, 0,
                gni_flags);
  if (err == -1) {
     fprintf(stderr, "getnameinfo() error: %s\n", gai_strerror(err));
     exit(EXIT_FAILURE);
  }
  printf("%ld bytes received from %s\n", bytes_recv, from_addr_str);

  /* Close socket and exit */
  err = close(sock_fd);
  if (err != 0) {
    perror("close() error");
    exit(EXIT_FAILURE);
  }

  return EXIT_SUCCESS;
}

/* Computes IPv4 checksum over the specified data */
uint16_t ipv4_checksum ( const void * data, size_t len )
{
  uint16_t word16;
  unsigned int sum = 0;
  const unsigned char * bytes = data;

  /* Make 16 bit words out of every two adjacent 8 bit words in the
   * packet and add them up */
```

```
 for (unsigned int i=0; i<len; i=i+2) {
  word16 = ((bytes[i] << 8) & 0xFF00) + (bytes[i+1] & 0x00FF);
  sum = sum + (unsigned int) word16;
 }

 /* Take only 16 bits out of the 32 bit sum and add up the carries */
 while (sum >> 16) {
  sum = (sum & 0xFFFF) + (sum >> 16);
 }

 /* One's complement the result */
 sum = ~sum;

 return (uint16_t) sum;
}
```

**Listado 6.2** – Información de uso de la aplicación ICMP de ejemplo (icmp)

```
$ ./icmp
Usage: icmp <dest>
 <dest>: Destination domain name or IP address
```

**Listado 6.3** – Ejecución de la aplicación ICMP de ejemplo (icmp)

```
$ sudo ./icmp www.example.com
ICMP Echo (64 bytes) sent to 93.184.216.34
84 bytes received from 93.184.216.34
```

El código del **Listado 6.1** es muy similar al del cliente UDP, puesto que los *sockets* "crudos" utilizan una semántica similar a los *sockets* de tipo datagrama, permitiendo a la aplicación tener un control total sobre cómo y cuándo se envían los mensajes. El único código más complejo está en la función ipv4_checksum(), que permite calcular el *checksum* del mensaje ICMP, aunque es el mismo algoritmo que usa IPv4, UDP y TCP. No hace falta entender este código, pero no es más que la suma en complemento a 1 de todos los octetos del mensaje cogidos de 2 en 2. Por eso los *checksums* de los protocolos IP son tan débiles, y la probabilidad de que un paquete IP corrupto tenga un *checksum* correcto no es nada despreciable.

Respecto a la ejecución de la aplicación, solo resaltar que ésta necesita ejecutarse con permisos de superusuario (e.g. utilizando el comando sudo como en el **Listado 6.3**), de forma que ésta pueda abrir el *socket* "crudo".

### 6.1.1. Principales llamadas al sistema de una aplicación con sockets "crudos"

Como ya hemos visto la mayoría de las llamadas al sistema en capítulos anteriores, vamos a centrarnos únicamente en las que presentan alguna diferencia:

- **int socket(int domain, int type, int protocol)** – Esta es la llamada fundamental de esta aplicación, porque crea un socket de tipo "crudo" (SOCK_RAW), indicando el protocolo que se quiere suplantar, en este caso ICMP (IPPROTO_ICMP), y por tanto debe utilizarse una familia IPv4 (AF_INET).

- **int setsockopt(int sockfd, int level, int optname, const void *optval, socklen_t optlen)** – Esta función permite configurar diferentes opciones de un *socket*. En el caso de Linux, los sockets "crudos" tienen la opción IP_HDRINCL, que permite indicar al núcleo si vamos a enviar únicamente la cabecera ICMP, o también la cabecera IPv4. En este caso, para simplificar el código de la aplicación, la hemos deshabilitado, pero si se activa permitiría enviar cabeceras IPv4 personalizadas (por ejemplo para generar fragmentos incorrectos, i.e. el ping de la muerte).

- **uint16_t htons(uint16_t hostshort)** – Aunque no es una llamada al sistema propiamente dicha, mientras se rellena la cabecera del mensaje ICMP se utiliza la función htons(). Esta familia de funciones hton*X*() se utilizan mucho en aplicaciones de red, porque la mayoría de los protocolos utilizan el denominado *network order* (i.e. *Little Endian*) para los campos de más de un octeto, mientras que hay muchas plataformas (incluyendo todos los Intel/AMD x86) que utilizan otro *host order* (i.e. *Big Endian*). Así que las funciones hton*X*() se utilizan para convertir números entre el *host* y el *network order* (solo si es necesario), mientras que la familia de funciones ntoh*X*() se utilizan para lo contrario. Las funciones htons() y ntohs() se utilizan para convertir números de 16 bits (*short* en C), pero también existen las funciones htonl() y ntohl() para convertir enteros de 32 bits (*long* en C).

- **ssize_t sendto(int sockfd, const void *buf, size_t len, int flags, const struct sockaddr *dest_addr, socklen_t addrlen)** - En la mayoría de los *sockets* "crudos" no tiene sentido realizar un connect() a un destino, por lo que habitualmente se utilizan las funciones sendto() y recvfrom() para indicar explícitamente el destino de cada paquete.

- **int poll(struct pollfd *fds, nfds_t nfds, int timeout)** – Al saltarnos capas de la torre de protocolos, tiene que ser la aplicación la que implemente los diferentes mecanismos de control de errores, flujo, congestión, etc. En este caso se utiliza el método poll() para implementar un temporizador de 1 segundo, de

forma que si el destino no responde, el programa salga, en lugar de quedarse bloqueado indefinidamente.

- **ssize_t recvfrom(int sockfd, void *buf, size_t len, int flags, struct sockaddr *src_addr, socklen_t *addrlen)** – Otra característica específica de los *sockets* "crudos" ICMP de Linux es que, aunque solo se envíen mensajes ICMP, al recibir también se obtiene la cabecera IPv4. Por eso en el **Listado 6.4** la aplicación indica que ha recibido 84 bytes = 64 bytes (mensaje ICMP) + 20 bytes (cabecera IPv4). Pero esto es un comportamiento específico de Linux que podría cambiar en otros sistemas operativos. Otro aspecto importante a tener en cuenta es que en muchos casos los paquetes que se reciben a través de un *socket* "crudo" también son procesados por la pila de protocolos (porque son paquetes válidos), y por tanto suele haber dos copias de cada paquete, uno el procesado por la pila de protocolos y otro el recibido por el *socket* crudo. Así que hay que tener cuidado para evitar que la pila de protocolos interfiera con la aplicación que usa el *socket* "crudo" (por ejemplo, si se está implementando el protocolo TCP a nivel de aplicación, cualquier segmento TCP recibido por la pila de protocolos del sistema podría generar un TCP RST, ya que ésta no es consciente de esa conexión). Una forma de evitar que la pila de protocolos procese esos paquetes es utilizando una dirección IP que no tenga configurada, y por tanto ignore los paquetes enviados a dicha dirección (aunque es necesario que la aplicación también implemente ARP o el descubrimiento de vecinos IPv6), o utilizar un identificador de protocolo IP no implementado (y desactivar o filtrar el envío de paquetes ICMP de Protocolo no alcanzable). No vale con utilizar un puerto TCP o UDP no utilizado, porque la pila de protocolos intentará notificar al origen que ese puerto no está en uso enviando paquetes TCP RST o ICMP Puerto no alcanzable. En este caso esto no es un problema, porque solo se deberían recibir mensajes ICMP de *Echo Reply*, y la pila de protocolos no debería responder a los mismos, aunque vayan a su dirección IP.

### 6.1.2. Utilización de 'strace' para analizar la aplicación ICMP

Del mismo modo que en capítulos anteriores, cuando solo se tiene acceso al ejecutable de la aplicación vamos a utilizar el comando 'strace' para analizar el comportamiento de la aplicación ICMP. Solo hay que comentar que para que la aplicación pueda abrir el *socket* "crudo" tiene que ejecutar como superusuario, por lo que debemos ejecutar el comando 'strace' con 'sudo'. También se podría lanzar 'sudo' desde 'strace', pero se verían todas las llamadas al sistema de 'sudo', complicando el análisis.

**Listado 6.4** – Traza de llamadas a sistema de la aplicación ICMP con 'strace' (simplificado)

```
$ sudo strace --trace=%net,%desc -tt -o icmp.log ./icmp www.example.com
ICMP Echo (64 bytes) sent to 93.184.216.34
84 bytes received from 93.184.216.34

$ cat /etc/resolv.conf
nameserver 127.0.0.53

$ cat icmp.log
11:25:58.565027 openat(AT_FDCWD, "/etc/ld.so.cache", O_RDONLY|O_CLOEXEC) = 3
11:25:58.565114 fstat(3, {st_mode=S_IFREG|0644, st_size=69936, ...}) = 0
11:25:58.565138 mmap(NULL, 69936, PROT_READ, MAP_PRIVATE, 3, 0)
            = 0x7fe1611db000
11:25:58.565158 close(3)              = 0
11:25:58.565179 openat(AT_FDCWD, "/lib/x86_64-linux-gnu/libc.so.6",
                  O_RDONLY|O_CLOEXEC) = 3
11:25:58.565201 read(3, "\177ELF\2\1\1\3\0\0\0\0\0\0\0\0\3\0>"..., 832) = 832
11:25:58.565220 pread64(3, "\6\0\0\0\4\0\0\0@\0\0\0\0\0\0"..., 784, 64) = 784
...
11:25:58.565274 fstat(3, {st_mode=S_IFREG|0755, st_size=2029224, ...}) = 0
11:25:58.565292 mmap(NULL, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|
                MAP_ANONYMOUS, -1, 0) = 0x7fe1611d9000
11:25:58.565315 pread64(3, "\6\0\0\0\4\0\0\0@\0\0\0\0\0\0"..., 784, 64) = 784
...
11:25:58.565402 mmap(NULL, 2036952, PROT_READ, MAP_PRIVATE|MAP_DENYWRITE, 3,
                0) = 0x7fe160fe7000
...
11:25:58.565528 close(3)              = 0


11:25:58.565768 openat(AT_FDCWD, "/etc/host.conf", O_RDONLY|O_CLOEXEC) = 3
11:25:58.565799 fstat(3, {st_mode=S_IFREG|0644, st_size=92, ...}) = 0
11:25:58.565822 read(3, "# The \"order\" line is only used "..., 4096) = 92
11:25:58.565846 read(3, "", 4096)      = 0
11:25:58.565863 close(3)              = 0
11:25:58.565880 openat(AT_FDCWD, "/etc/resolv.conf", O_RDONLY|O_CLOEXEC) = 3
11:25:58.565905 fstat(3, {st_mode=S_IFREG|0644, st_size=717, ...}) = 0
11:25:58.565923 read(3, "# This file is managed by man:sy"..., 4096) = 717
11:25:58.565945 read(3, "", 4096)      = 0
11:25:58.565961 close(3)              = 0
11:25:58.565991 socket(AF_UNIX, SOCK_STREAM|SOCK_CLOEXEC|SOCK_NONBLOCK, 0) = 3
11:25:58.566014 connect(3, {sa_family=AF_UNIX,
                  sun_path="/var/run/nscd/socket"}, 110)
            = -1 ENOENT (No existe el archivo o el directorio)
11:25:58.566142 close(3)              = 0
11:25:58.566161 socket(AF_UNIX, SOCK_STREAM|SOCK_CLOEXEC|SOCK_NONBLOCK, 0) = 3
11:25:58.566179 connect(3, {sa_family=AF_UNIX,
                  sun_path="/var/run/nscd/socket"}, 110)
            = -1 ENOENT (No existe el archivo o el directorio)
11:25:58.566201 close(3)              = 0
11:25:58.566219 openat(AT_FDCWD, "/etc/nsswitch.conf", O_RDONLY|O_CLOEXEC) = 3
11:25:58.566243 fstat(3, {st_mode=S_IFREG|0644, st_size=542, ...}) = 0
```

```
11:25:58.566264 read(3, "# /etc/nsswitch.conf\n#\n# Example"..., 4096) = 542
11:25:58.566286 read(3, "", 4096)       = 0
11:25:58.566302 close(3)                = 0
11:25:58.566344 openat(AT_FDCWD, "/etc/ld.so.cache", O_RDONLY|O_CLOEXEC) = 3
11:25:58.566375 fstat(3, {st_mode=S_IFREG|0644, st_size=69936, ...}) = 0
11:25:58.566395 mmap(NULL, 69936, PROT_READ, MAP_PRIVATE, 3, 0)
            = 0x7fe1611db000
11:25:58.566414 close(3)                = 0
11:25:58.566434 openat(AT_FDCWD, "/lib/x86_64-linux-gnu/libnss_files.so.2",
                  O_RDONLY|O_CLOEXEC) = 3
11:25:58.566456 read(3, "\177ELF\2\1\1\0\0\0\0\0\0\0\0\0\3\0>"..., 832) = 832
11:25:58.566475 fstat(3, {st_mode=S_IFREG|0644, st_size=51832, ...}) = 0
11:25:58.566494 mmap(NULL, 79672, PROT_READ, MAP_PRIVATE|MAP_DENYWRITE, 3, 0)
            = 0x7fe160fd3000
...
11:25:58.566613 close(3)                = 0
11:25:58.566680 openat(AT_FDCWD, "/etc/hosts", O_RDONLY|O_CLOEXEC) = 3
11:25:58.566708 lseek(3, 0, SEEK_CUR)   = 0
11:25:58.566725 fstat(3, {st_mode=S_IFREG|0644, st_size=229, ...}) = 0
11:25:58.566744 read(3, "127.0.0.1\tlocalhost\n127.0.1.1\tub"..., 4096) = 229
11:25:58.566765 lseek(3, 0, SEEK_CUR)   = 229
11:25:58.566780 read(3, "", 4096)       = 0
11:25:58.566796 close(3)                = 0
11:25:58.566814 openat(AT_FDCWD, "/etc/ld.so.cache", O_RDONLY|O_CLOEXEC) = 3
11:25:58.566835 fstat(3, {st_mode=S_IFREG|0644, st_size=69936, ...}) = 0
11:25:58.566853 mmap(NULL, 69936, PROT_READ, MAP_PRIVATE, 3, 0)
            = 0x7fe1611db000
11:25:58.566870 close(3)                = 0
11:25:58.566889 openat(AT_FDCWD,
                  "/lib/x86_64-linux-gnu/libnss_mdns4_minimal.so.2",
                  O_RDONLY|O_CLOEXEC) = 3
11:25:58.566910 read(3, "\177ELF\2\1\1\0\0\0\0\0\0\0\0\0\3\0>"..., 832) = 832
11:25:58.566929 fstat(3, {st_mode=S_IFREG|0644, st_size=18504, ...}) = 0
11:25:58.566947 mmap(NULL, 20496, PROT_READ, MAP_PRIVATE|MAP_DENYWRITE, 3, 0)
            = 0x7fe160fcd000
...
11:25:58.567057 close(3)                = 0
11:25:58.567079 openat(AT_FDCWD, "/lib/x86_64-linux-gnu/libresolv.so.2",
                  O_RDONLY|O_CLOEXEC) = 3
11:25:58.567101 read(3, "\177ELF\2\1\1\0\0\0\0\0\0\0\0\0\3\0>"..., 832) = 832
11:25:58.567120 fstat(3, {st_mode=S_IFREG|0644, st_size=101320, ...}) = 0
11:25:58.567139 mmap(NULL, 113280, PROT_READ, MAP_PRIVATE|MAP_DENYWRITE, 3, 0)
            = 0x7fe160fb1000
...
11:25:58.567262 close(3)                = 0
11:25:58.567372 openat(AT_FDCWD, "/etc/ld.so.cache", O_RDONLY|O_CLOEXEC) = 3
11:25:58.567400 fstat(3, {st_mode=S_IFREG|0644, st_size=69936, ...}) = 0
11:25:58.567419 mmap(NULL, 69936, PROT_READ, MAP_PRIVATE, 3, 0)
            = 0x7fe1611db000
11:25:58.567437 close(3)                = 0
11:25:58.567457 openat(AT_FDCWD, "/lib/x86_64-linux-gnu/libnss_dns.so.2",
                  O_RDONLY|O_CLOEXEC) = 3
11:25:58.567479 read(3, "\177ELF\2\1\1\0\0\0\0\0\0\0\0\0\3\0>"..., 832) = 832
11:25:58.567498 fstat(3, {st_mode=S_IFREG|0644, st_size=31176, ...}) = 0
```

```
11:25:58.567517 mmap(NULL, 32984, PROT_READ, MAP_PRIVATE|MAP_DENYWRITE, 3, 0)
            = 0x7fe160fa8000
...
11:25:58.567601 close(3)              = 0
11:25:58.567668 socket(AF_INET, SOCK_DGRAM|SOCK_CLOEXEC|SOCK_NONBLOCK,
                  IPPROTO_IP) = 3
11:25:58.567696 setsockopt(3, SOL_IP, IP_RECVERR, [1], 4) = 0
11:25:58.567718 connect(3, {sa_family=AF_INET, sin_port=htons(53),
                   sin_addr=inet_addr("127.0.0.53")}, 16) = 0
11:25:58.567750 poll([{fd=3, events=POLLOUT}], 1, 0)
            = 1 ([{fd=3, revents=POLLOUT}])
11:25:58.567772 sendto(3, "U\3\1 \0\1\0\0\0\0\0\1\3www\7example\3com\0"...,
                  44, MSG_NOSIGNAL, NULL, 0) = 44
11:25:58.567951 poll([{fd=3, events=POLLIN}], 1, 5000)
            = 1 ([{fd=3, revents=POLLIN}])
11:25:58.567980 ioctl(3, FIONREAD, [60]) = 0
11:25:58.568001 recvfrom(3, "U\3\201\200\0\1\0\1\0\0\0\1\3www\7example\3"...,
                   1024, 0, {sa_family=AF_INET, sin_port=htons(53),
                   sin_addr=inet_addr("127.0.0.53")}, [28->16]) = 60
11:25:58.568039 close(3)              = 0


11:25:58.568062 socket(AF_INET, SOCK_RAW, IPPROTO_ICMP) = 3
11:25:58.568085 setsockopt(3, SOL_IP, IP_HDRINCL, [0], 4) = 0
11:25:58.568105 sendto(3, "\10\0\355\264\0237\0\1\0\1\2\3\4\5\6\7\10\t\n"...,
                  64, 0, {sa_family=AF_INET, sin_port=htons(0),
                  sin_addr=inet_addr("93.184.216.34")}, 16) = 64
11:25:58.568141 fstat(1, {st_mode=S_IFCHR|0620, st_rdev=makedev(0x88, 0),
                 ...}) = 0
11:25:58.568163 write(1, "ICMP Echo (64 bytes) sent to 93."..., 43) = 43
11:25:58.568278 poll([{fd=3, events=POLLIN|POLLERR}], 1, 3000)
            = 1 ([{fd=3, revents=POLLIN}])
11:25:58.686663 recvfrom(3, "E\0\0T8\254\0\0005\1UY]\270\330\"\300\250\1!"...,
                   1480, 0, {sa_family=AF_INET, sin_port=htons(0),
                   sin_addr=inet_addr("93.184.216.34")}, [16]) = 84
11:25:58.686732 write(1, "84 bytes received from 93.184.21"..., 37) = 37
11:25:58.686872 close(3)              = 0
11:25:58.687076 +++ exited with 0 +++
```

Al utilizar un nombre DNS como dirección destino de la aplicación ICMP, el número de llamadas que aparece al principio del **Listado 6.4** es bastante más elevado que en trazas anteriores, e incluso aparece un socket UDP (11:25:58.567668), aunque este se utiliza únicamente para preguntar por el nombre DNS 'www.example.com' al servidor DNS local (127.0.0.53:UDP/53). Por lo tanto, todas estas llamadas son el resultado de llamar a la función getaddrinfo().

Las llamadas al sistema que aparecen al final del **Listado 6.4** ya son las que aparecen tan cual en el código fuente de la aplicación (**Listado 6.1**).

## 6.2. Análisis del tráfico de la aplicación ICMP de ejemplo

Una vez analizadas las funciones de red que utiliza la aplicación ICMP de ejemplo, vamos a analizar el tráfico de red que genera.

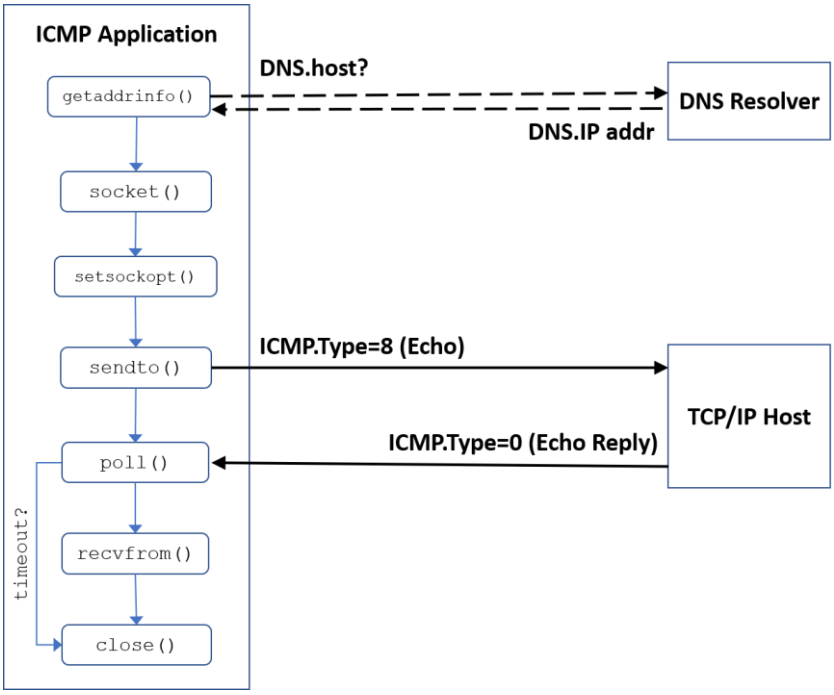



**Figura 6.5** – Modelo de la aplicación ICMP de ejemplo

La **Figura 6.5** muestra un modelo simplificado del flujo de ejecución de la aplicación ICMP de ejemplo, así como el tráfico que genera. De nuevo, el flujo es muy similar al del cliente UDP, debido a los *sockets* "crudos" se comportan de una forma similar a los *sockets* de tipo datagrama, como los *sockets* UDP.

Del lado destino no se muestra el flujo interno del TCP/IP Host, porque normalmente el procesamiento de los paquetes ICMP se realiza dentro del núcleo del sistema operativo. Aunque no es tan habitual (por los problemas de que la pila de protocolos del sistema también recibe las peticiones de los clientes), también se podría implementar una aplicación de tipo servidor que utilice *sockets* "crudos". Tendría un flujo similar al del servidor UDP, o al de esta aplicación ICMP revirtiendo el orden del sendto() y el recvfrom() y eliminando el poll().

**Listado 6.6** – Configuración y ejecución de la aplicación ICMP en el equipo 192.168.1.33

```
muruenya@192.168.1.33:~$ route -n
Tabla de rutas IP del núcleo
Destino         Pasarela        Genmask         Indic Métric Ref    Uso Interfaz
0.0.0.0         192.168.1.1     0.0.0.0         UG    100    0        0 eth0
192.168.1.0     0.0.0.0         255.255.255.0   U     100    0        0 eth0

muruenya@192.168.1.33:~$ cat /etc/resolv.conf
```

```
nameserver 127.0.0.53

muruenya@192.168.1.33:~$ resolvectl dns eth0
Link 2 (eth0): 80.58.61.250 80.58.61.254

muruenya@192.168.1.33:~$ sudo ./icmp www.example.com
ICMP Echo (64 bytes) sent to 93.184.216.34
84 bytes received from 93.184.216.34
```

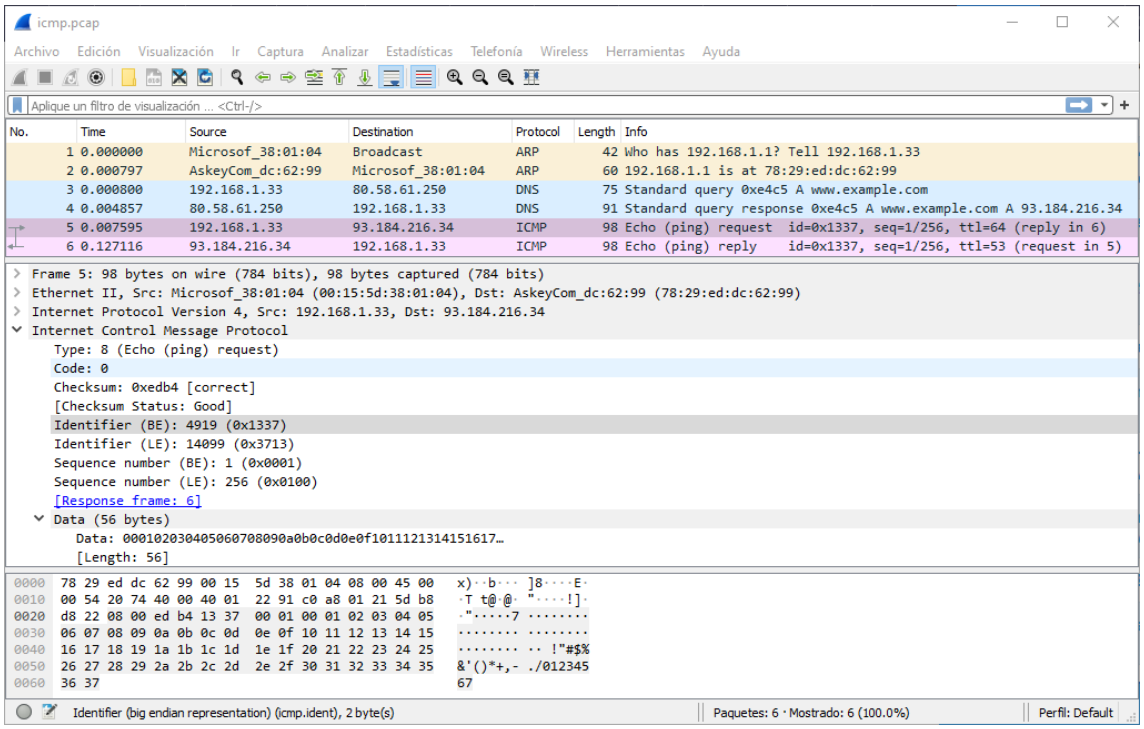



**Figura 6.7** – Captura de tráfico de la aplicación ICMP de ejemplo (icmp.pcap)

Pasemos ahora a analizar el tráfico de red generado por esta aplicación. La **Figura 6.7** muestra una captura de tráfico de red realizada desde el equipo 192.168.1.33 (sudo tcpdump -i eth0 -s0 -w icmp.pcap), desde donde se lanza la aplicación ICMP.

Como a la aplicación ICMP se le pasa como destino el nombre DNS 'www.example.com', la función getaddrinfo() debe realizar una consulta al servidor DNS configurado. Aunque no se aprecia en el fichero de captura, porque ésta se ha realizado únicamente en el interfaz de red externo (eth0), realmente primero se realiza una consulta DNS al servidor local 127.0.0.53 (ver **Listado 6.6**), pero como no la tiene cacheada, éste reenvía la consulta al servidor DNS del operador de servicio de Internet (80.58.61.250). Como ese servidor está fuera de la subred, el equipo cliente debe reenviar los datagramas al *router* por defecto (192.168.1.1), por eso el primer intercambio ARP en el que se pregunta por su dirección MAC (78:29:ed:dc:62:99), dado que (extrañamente) el cliente tampoco tiene en su cache

ARP la dirección del *router* por defecto. Nótese que esa también es la dirección MAC destino a la que se envía el paquete ICMP Echo generado por la aplicación.

Por tanto, el primer paquete realmente generado por la aplicación ICMP es el seleccionado en 'Wireshark' (#5) que, a pesar de haber sido generado a nivel de aplicación (ver Identifier (BE) = 0x1337), 'Wireshark' lo interpreta como un paquete ICMP válido (i.e. ver campo *checksum*). Un aspecto interesante del interfaz de 'Wireshark' es que los valores de los campos *Identifier* y *Sequence number* (que tienen 16 bits de largo) se muestran tanto en formato *Big Endian* (BE) como *Little Endian* (LE). Aunque formalmente todos los protocolos TCP/IP definidos por el IETF utilizan en denominado *Network Order* (i.e. *Big Endian*), y por eso el código fuente de la aplicación ICMP utiliza la función htons() para convertir los valores de 16 bits a ese formato, hay aplicaciones que no realizan esa conversión, y por tanto envían esos campos en formato *Little Endian*, porque esos campos solo tienen sentido para el emisor del mensaje ICMP Echo y el equipo receptor solo debe devolverlos tal cual los recibe. Por lo tanto, 'Wireshark' muestra ambos valores, para que se pueda interpretar correctamente el número de secuencia de los paquetes. Pero la mejor prueba que el paquete ICMP generado por la aplicación es un mensaje ICMP valido, es que el destino genera una respuesta ICMP *Echo Reply* (paquete #6), con el mismo identificador y número de secuencia que el enviado en el ICMP Echo.

Además, si suponemos que el destino mandó el mensaje ICMP *Echo Reply* con un TTL=64, que es el valor más habitual que utilizan los sistemas operativos modernos, al llegarnos con un TTL=53, quiere decir que hay unos 11 saltos entre nuestro equipo y el destino.

Podemos comprobar esta hipótesis lanzando un comando 'traceroute' al destino (**Listado 6.8**), que muestra que hay 12 saltos en el sentido 192.168.1.33 → 93.184.216.34, por lo que sí es probable que haya 11 en el sentido inverso.

**Listado 6.8** – Ejecución del comando 'traceroute' en el equipo 192.168.1.33

```
muruenya@192.168.1.33:~$ traceroute --icmp -q1 -n www.example.com
traceroute to www.example.com (93.184.216.34), 30 hops max, 60 byte packets
 1  192.168.1.1  1.004 ms
 2  192.168.144.1  3.949 ms
 3  81.46.66.201  3.947 ms
 4  *
 5  *
 6  213.140.51.58  5.405 ms
 7  *
 8  *
 9  *
10  84.16.9.107  126.560 ms
11  152.195.88.141  129.118 ms
12  93.184.216.34  120.684 ms
```

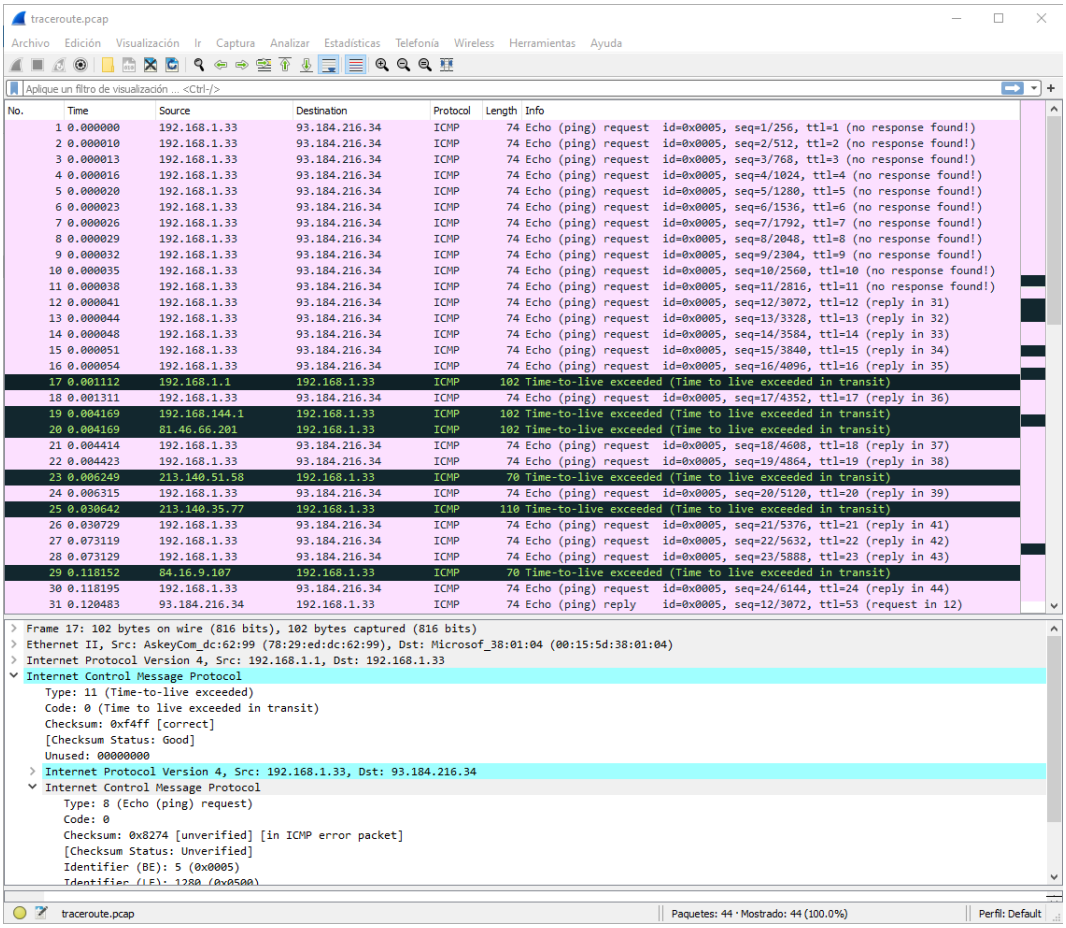



**Figura 6.9** – Captura de tráfico del comando 'traceroute' (traceroute.pcap)

Podemos entender el funcionamiento del comando 'traceroute' observando su tráfico (**Figura 6.9**). Básicamente va enviando mensajes ICMP *Echo* con tiempos de vida que van de TTL=1 (paquete #1) a TTL=24 (paquete #30), hasta recibir un *Echo Reply* (#31) del destino (93.184.216.34) como respuesta al ICMP Echo que se envió con TTL=12. Las direcciones de los *routers* intermedios se obtienen de los mensajes ICMP de *Time Exceeded* que envían los *routers* intermedios cuando reciben un datagrama IPv4 con TTL=1. Y se puede saber a qué mensaje ICMP *Echo* corresponde, porque los mensajes ICMP *Time Exceeded* incluyen la cabecera IPv4 y la cabecera ICMP del datagrama que generó dicho error, tal como se puede observar en el paquete (#17) seleccionado en 'Wireshark', que genera el *router* por defecto de la subred (192.168.1.1) al procesar el primer mensaje *Echo* con TTL=1 (número de secuencia 1).

En el **Listado 6.8** hay varios saltos de los que no se tiene información (i.e. en lugar de la dirección IP del *router* intermedio solo aparece un asterisco), ya que no se ha recibido ningún mensaje ICMP *Time Exceeded* a los mensajes con ese TTL. Esto se puede deber a una multitud de factores, como por ejemplo si algún *firewall* intermedio filtra el tráfico ICMP (esto es muy habitual en los últimos saltos, ya que

las organizaciones quieren ocultar su topología interna), pero otra causa muy habitual es que la generación de tráfico ICMP está limitada en muchos dispositivos (especialmente en los de red). Por esa razón, además de las posibles pérdidas y para detectar múltiples rutas, 'traceroute' por defecto envía 3 mensajes por cada salto. Aunque en este caso lo hemos limitado a 1 (con la opción `-q1`) para que la captura fuera más simple.

# 7. Captura y Análisis de Tráfico TLS/SSL

Tal como se comentó en el **Capítulo 2**, las versiones modernas de TLS sin *ciphersuites* débiles se consideran seguras, por lo que no es posible descifrar su contenido a no ser que se cuente con la colaboración de uno de los extremos de la comunicación, y en algunos casos también es necesario realizar un ataque activo de *Adversary-in-the-Middle* (MitM).

## 7.1. Descifrado con la clave RSA privada del servidor

El método pasivo más "clásico" para descifrar el contenido de una sesión SSL consistía en utilizar la clave RSA privada del servidor, puesto que en las primeras versiones de SSL el cliente envía el *Pre-Master Secret*, que se utiliza para generar todas las claves simétricas de la sesión, al servidor cifrado con su clave RSA pública, por lo que si se dispone de la clave privada asociada, se puede descifrar y por tanto derivar todas las claves necesarias para descifrar los dos sentidos de la sesión SSL, incluso si no se disponía de ella en el momento de realizar la captura.

Lamentablemente (desde el punto de vista del analista, afortunadamente desde el punto de vista de la seguridad), las versiones modernas de TLS suelen utilizar un mecanismo de intercambio de claves basado en Diffie-Hellman Efímero (DHE) o basado en Curvas Elípticas (ECDH), lo que impide derivar la claves, incluso se dispone de la clave privada, puesto que eso requeriría romper el intercambio de claves DH, lo cual no es computacionalmente posible a no ser que se utilicen grupos DH extremadamente débiles. 'Wireshark' permite configurar (en "Edición > Preferencias.. > *Protocols* > TLS") la clave RSA privada de un servidor (**Figura 7.1**), para así descifrar las sesiones SSL basadas en el intercambio de claves mediante RSA.

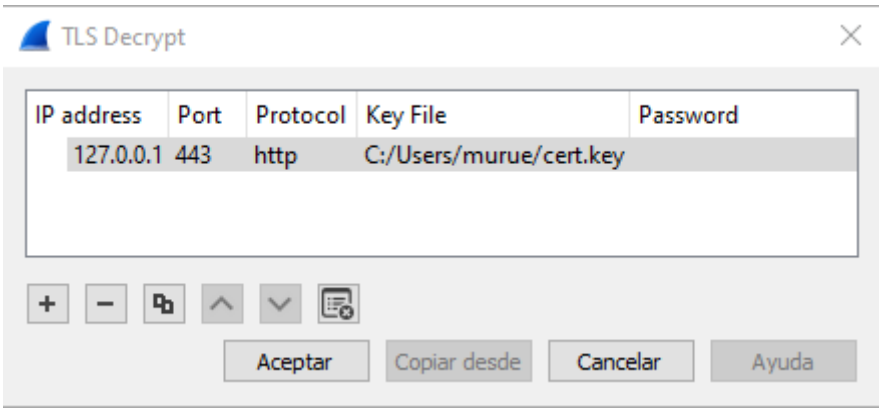



**Figura 7.1** – Lista de claves privadas RSA en 'Wireshark'

Aunque, si se dispone de la clave privada del servidor, es posible suplantarlo a todos los efectos, puesto que podemos demostrar que somos los propietarios de su

certificado X.509. Así que la alternativa moderna para descifrar el tráfico de un servidor, cuando se dispone de su clave privada, consiste en montar un *proxy* inverso, configurarle el certificado y la clave privada del servidor original, y conseguir desviar el tráfico destinado al servidor al *proxy*, por ejemplo, modificando la resolución DNS de los clientes (e.g. /etc/hosts).

De esta forma, el cliente trata al *proxy* como si fuera el servidor TLS original (porque éste también es capaz de demostrar la posesión de su certificado), lo que le hace establecer el *handshake* TLS con él, incluido el intercambio de claves, así que da igual si se realiza con la clave RSA del servidor o mediante DHE o ECDH, porque el *proxy* dispondrá de todas las claves de la sesión y podrá descifrar toda la información intercambiada, que luego enviará al servidor TLS original, haciéndose pasar por el cliente y completando así el ataque de AitM activo. En el **Apartado 7.3** explicaremos en más detalle una variante de este ataque.

Sin embargo, la clave privada de un servidor TLS es literalmente el secreto más importante del mismo, por lo que si se dispone del mismo se puede suponer que se dispone un control total sobre él y en ese caso ha formas más sencillas de analizarlo, dado que los administradores son (con mucha razón) muy reticentes a entregar la clave privada de sus servidores a un analista.

## 7.2. Descifrando con las claves de la sesión TLS

Si no se dispone de la clave privada del servidor, la otra alternativa para realizar un descifrado pasivo consiste en contar con la colaboración de la aplicación cliente. En particular hay algunas aplicaciones, como el comando 'curl', la herramienta 'mitmproxy' o los navegadores Firefox y Chrome, que pueden configurarse para que almacenen en un fichero el *Pre-Master Secret* de todas las sesiones TLS que negocian. Para ello basta con definir la variable de entorno SSLKEYLOGFILE apuntando al fichero donde se quiere almacenar los *Pre-Master Secret*. Luego, hay que indicar a 'Wireshark' (en "Edición > Preferencias... > *Protocols* > TLS") dónde se encuentra el fichero (**Figura 7.2**).

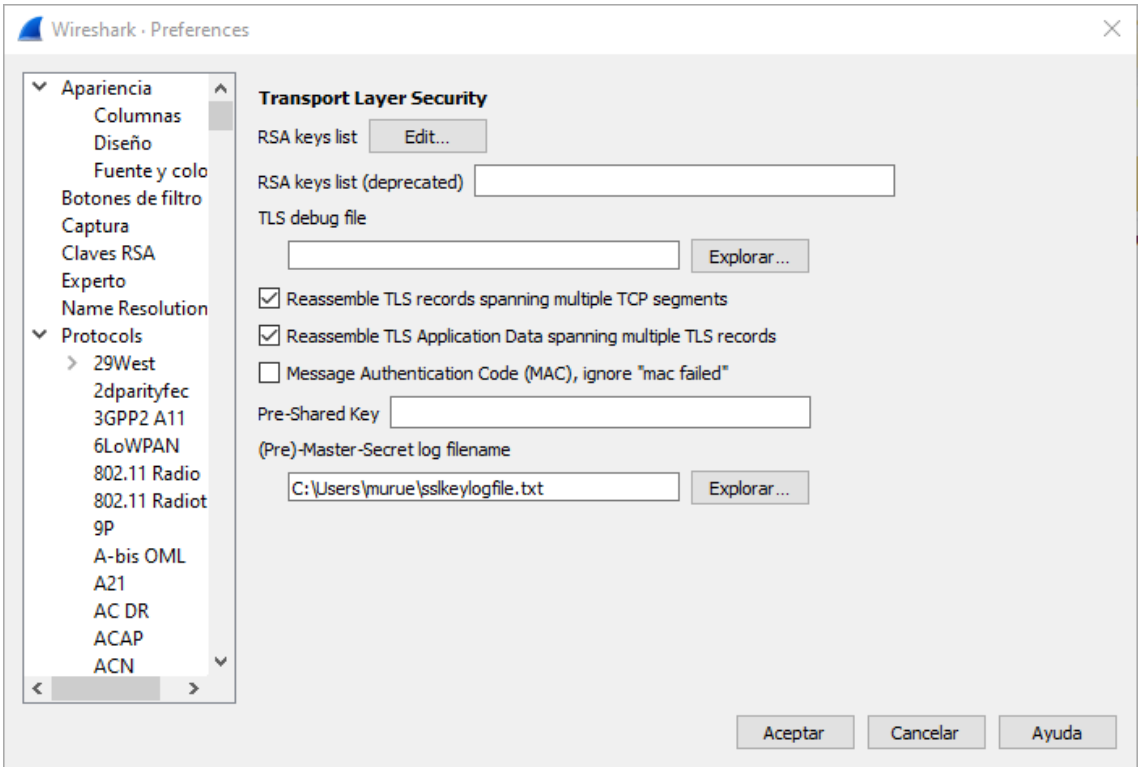



**Figura 7.2** – Opciones de descifrado de sesiones TLS con 'Wireshark'

Una vez hecho esto, si se lanza un comando que soporte esta variable de entorno (**Listado 7.3**), almacenará en el fichero indicado los secretos utilizados en la sesión TLS (que en el fichero se identifica por el valor del *ClientSecret*), y 'Wireshark' será capaz de descifrar los datos capturados de la sesión (**Figura 7.4**).

**Listado 7.3** – Volcado de claves de una sesión TLS con SSLKEYLOGFILE

```
$ export SSLKEYLOGFILE=/home/muruenya/sslkeylogfile.txt

$ curl https://www.example.com
<!doctype html>
<html>
<head>
  <title>Example Domain</title>

  <meta charset="utf-8" />
  <meta http-equiv="Content-type" content="text/html; charset=utf-8" />
  <meta name="viewport" content="width=device-width, initial-scale=1" />
...

$ cat sslkeylogfile.txt
SERVER_HANDSHAKE_TRAFFIC_SECRET
06fa3511a214d0f81b8e0c7586fc087084001773668f11747fb5d84df61de190
9500854084e33d5ba09ed3b9e89be08a53d6bf4893be2d55440b9f993f3029e0c52b8d78f82484e31c5
d28c3c5554e9b
...
```

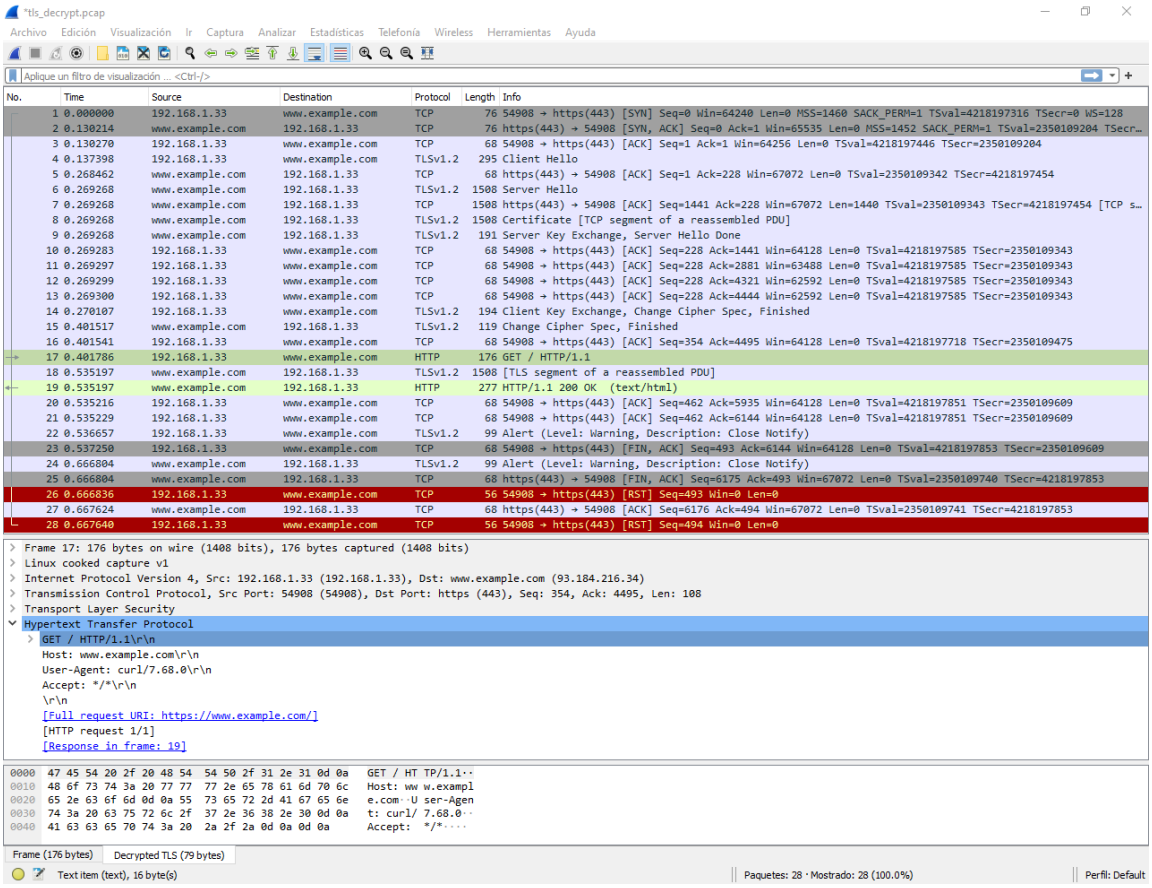



**Figura 7.4** – Descifrando una sesión TLS con 'Wireshark'

## 7.3. Realizando un ataque de *Adversary-in-the-Middle* a TLS

Obviamente la técnica anterior de descifrado pasivo solo es posible si se usa una de las aplicaciones que permiten almacenar las claves de sesión TLS, y de hecho la mayoría de las que lo hacen proporcionan mecanismos más simples para analizar el contenido de las comunicaciones (e.g. 'curl -v' proporciona información detallada sobre la comunicación, y las herramientas de desarrollo de Chrome y Firefox también permiten analizar todo su tráfico).

Por lo tanto, la técnica más habitual para descifrar tráfico TLS es mediante un ataque activo de *Adversary-in-the-Middle* (AitM), que solo es posible si se dispone de la clave privada del servidor TLS legítimo (que ya analizamos en el **Apartado 7.1**), o consiguiendo que el cliente confíe en una Autoridad de Certificación (CA) bajo nuestro control, y por lo tanto podamos emitir un certificado para el servidor destino del que se fíe el cliente. Pero es muy importante resaltar ese requisito: debemos ser capaces de poder controlar la lista de autoridades de certificación confiables de la aplicación origen, para que este ataque pueda tener éxito (porque TLS se diseñó literalmente para evitar este tipo de ataques).

Si no tenemos ningún control sobre el sistema donde ejecuta la aplicación cliente, la única alternativa restante es lanzar aun así el ataque de AitM y confiar en que la aplicación no ha implementado correctamente la verificación de la cadena de certificados del servidor, de forma que le valga cualquier certificado con el nombre del servidor, independientemente de la autoridad de certificación que lo haya emitido (o incluso si es un certificado auto-firmado). Porque sorprendentemente hay un gran número de aplicaciones que, debido a la supuesta complejidad de la gestión de certificados, solo validan el certificado del servidor de manera opcional, o simplemente muestran una advertencia al usuario y/o registran en un log el uso de un certificado invalido, pero continúan con la comunicación.

La herramienta 'mitmproxy', que describimos en el **Apartado 3.2.3**, además de trabajar como proxy HTTP directo e inverso, tiene la capacidad interceptar tráfico cifrado con TLS, automatizando la generación de certificados de manera dinámica en función del servidor al que quiera acceder el cliente.

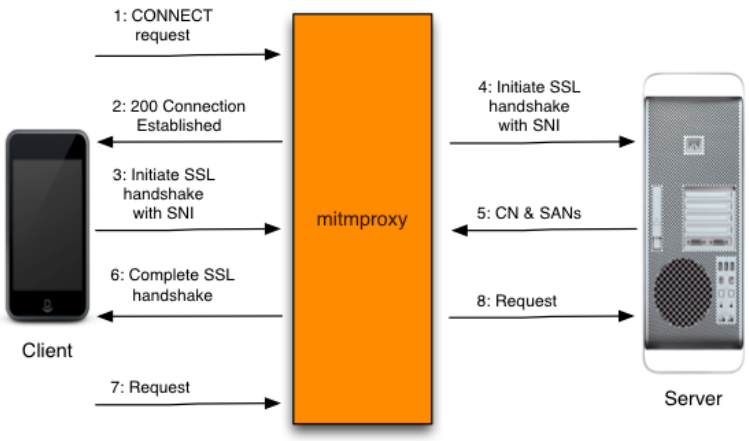



**Figura 7.5** – Interceptación de sesiones TLS con 'mitmproxy' [Fuente]

Para entender su funcionamiento podemos basarnos en la **Figura 7.5**. En ella el cliente de la sesión TLS utiliza 'mitmproxy' como su proxy HTTPS explícito (como ya hemos dicho, 'mitmproxy' también soporta la interceptación en modo transparente o como *proxy* inverso, pero para comprender su funcionamiento usaremos el escenario de captura más sencillo). Así que cuando quiere conectarse a un sitio HTTP envía una petición CONNECT indicando el nombre DNS o dirección IP del servidor destino (#1). Un *proxy* HTTPS convencional establecería una conexión TCP con el servidor destino, y a partir de ese momento reenviaría todo el tráfico entre el cliente y el servidor de manera transparente, por lo que la sesión TLS se negociaría y establecería entre el cliente y servidor, así que el *proxy* solo vería los mensajes TLS cifrados.

'mitmproxy' se comporta de manera diferente. Primero responde al cliente con un 200 OK (#2) indicando (falsamente) que ya se ha conectado al servidor, para que el

cliente le mande el mensaje TLS de Client Hello (#3), de forma que pueda utilizar el valor del *Server Name Indication* (SNI) para identificar el nombre DNS del servidor TLS al que desea conectarse el cliente. Por lo que no es hasta ese momento cuando el *proxy* establece una sesión TLS con el servidor (#4). 'mitmproxy' utiliza el nombre común (CN) y los *Subject Alternative Name* (SAN) del certificado que le envía el servidor (#5) para generar un certificado similar con ellos (con una clave RSA generada por el *proxy* y firmado con su propia CA) y enviárselo al cliente. Si el cliente tiene a la CA del 'mitmproxy' en su lista de CAs de confianza, se fiará de ese certificado y completará la conexión TLS con el 'mitmproxy' (#6), suponiendo que éste es el servidor final. En ese momento habrá dos sesiones TLS establecidas: una entre el cliente y el 'mitmproxy', y otra entre el 'mitmproxy' y el servidor, por lo que el 'mitmproxy' podrá descifrar el tráfico de ambas y reenviar los datos entre ellas (#7-8).

Por tanto, para que el ataque funcione es imprescindible que antes añadamos una CA bajo nuestro control en la lista de autoridades de certificación del cliente. La primera vez que ejecuta 'mitmproxy' (**Listado 7.6**) se genera un certificado auto-firmado (en diversos formatos para que sea fácil de instalar en diferentes sistemas operativos y librerías) que actuará como una CA para generar dinámicamente los certificados que utiliza 'mitmproxy' para suplantar al servidor.

**Listado 7.6** – Creación de la Autoridad de Certificación de 'mitmproxy' en su primera ejecución

```
$ mitmproxy

$ ls -a ~/.mitmproxy
.  mitmproxy-ca-cert.cer  mitmproxy-ca-cert.pem  mitmproxy-ca.pem
.. mitmproxy-ca-cert.p12  mitmproxy-ca.p12      mitmproxy-dhparam.pem
```

Ahora hay que instalar el certificado en la lista de CAs del sistema cliente, aunque como TLS normalmente se implementa a nivel de aplicación con alguna librería criptográfica (e.g. OpenSSL), el procedimiento para añadir una CA de confianza depende mucho de la aplicación o sistema.

En Windows sí hay un repositorio centralizado de certificados y CAs de confianza. Para añadir el certificado del 'mitmproxy' se puede utilizar el comando 'certutil' desde una línea de comandos con permisos de administración (**Listado 7.7**), o mediante la herramienta gráfica 'certmgr.msc' (**Figura 7.8**).

**Listado 7.7** – Instalación de una Autoridad de Certificación en Windows con 'certutil'

```
> certutil -addstore root mitmproxy-ca-cert.cer
root "Entidades de certificación raíz de confianza"
La firma coincide con la clave pública
El certificado "mitmproxy" se ha agregado al almacén.
CertUtil: -addstore comando completado correctamente.
```

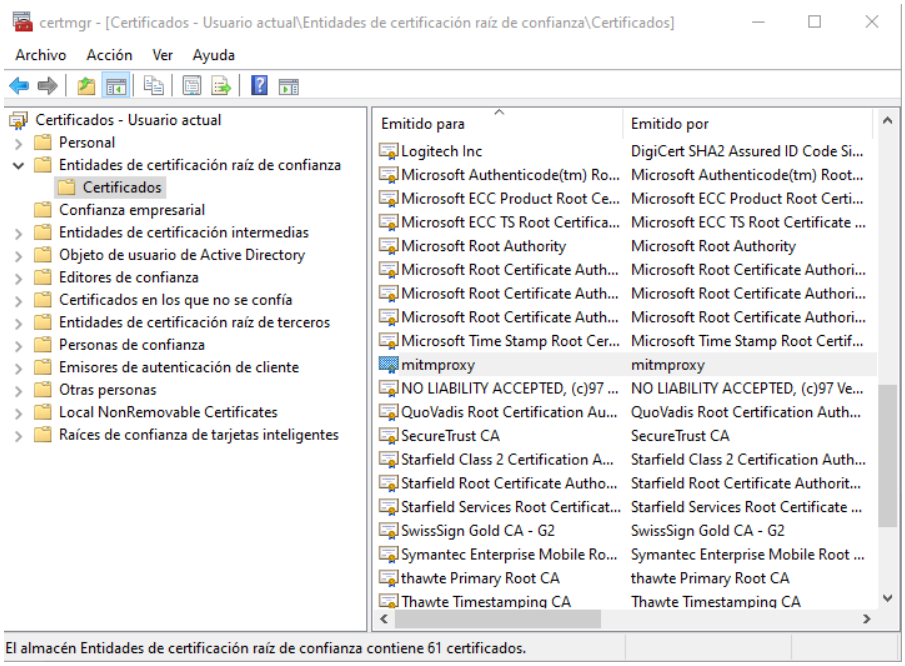



**Figura 7.8** – Lista de Autoridades de Certificación de Confianza de Windows

La mayoría de las aplicaciones Windows se basan en la lista de CAs del sistema operativo, incluyendo navegadores como Edge, Internet Explorer y Chrome. Sin embargo, hay ciertas aplicaciones como las basadas en Java (**Listado 7.9**), o navegadores como Firefox (**Figura 7.10**) que tienen su propio repositorio de certificados.

**Listado 7.9** – Importación de Autoridad de Certificación de 'mitmproxy' en Java

```
$ sudo keytool -importcert -alias mitmproxy -storepass changeit -keystore
-cacerts -trustcacerts -file ~/.mitmproxy/mitmproxy-ca-cert.pem
Propietario: O=mitmproxy, CN=mitmproxy
Emisor: O=mitmproxy, CN=mitmproxy
Número de serie: ea49ead9211
...
¿Confiar en este certificado? [no]:  si
Se ha agregado el certificado al almacén de claves
```

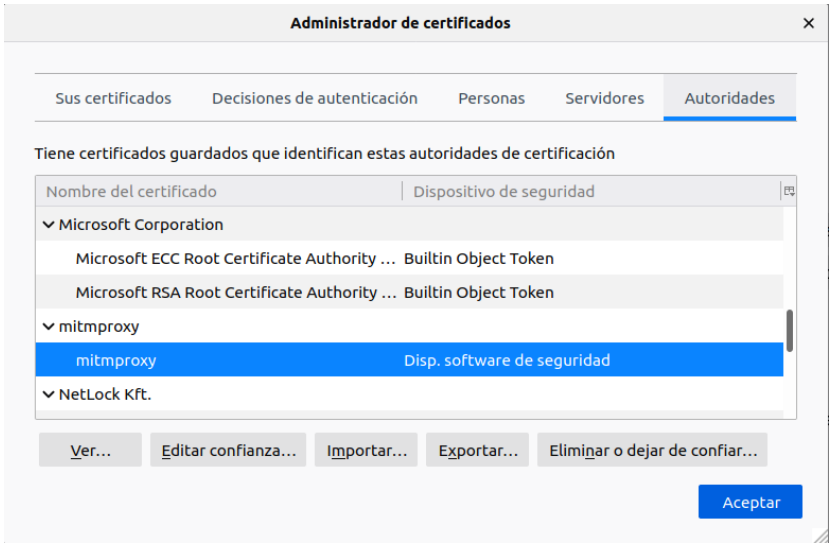



**Figura 7.10** – Administrador de certificados de Firefox

Una vez instalado el certificado en el sistema cliente, basta con arrancar el *proxy*, sin necesidad de especificar el certificado de la CA, puesto que está en el directorio por defecto (**Listado 7.11**).

**Listado 7.11** – Ejecución de la herramienta 'mitmproxy' en modo *proxy* HTTPS.

```
$ mitmproxy --mode regular --listen-host 127.0.0.1 --listen-port 8080
```

Para trabajar en modo *proxy* inverso HTTPS (útil para el ataque de AitM con el certificado del servidor que se comentó en el **Apartado 7.1**), hay que crear un fichero PEM que incluya tanto el certificado del servidor suplantado y el de la autoridad de certificación intermedia que lo han emitido, como la clave privada del mismo. Luego hay que arrancar 'mitmproxy' en modo *proxy* inverso (**Listado 7.12**), indicando el nombre del fichero y su certificado+clave. Como habrá que escuchar en el puerto 443 (i.e. https), que es un puerto bien conocido, es necesario lanzar el *proxy* con permisos de superusuario.

**Listado 7.12** – Ejecución de la herramienta 'mitmproxy' en modo *proxy* inverso.

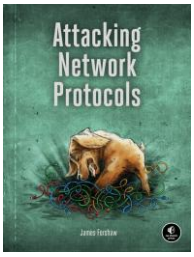
```
$ openssl genra -out cert.key 2048

$ openssl req -new -key cert.key -subj "/C=ES/ST=Madrid/O=mitmproxy/
CN=www.example.com" -out cert.csr

$ openssl x509 -req -days 365 -in cert.csr -signkey ca.key -out cert.crt
Signature ok
Subject=C = ES, ST = Madrid, O = mitmproxy, CN = www.example.com
Getting Private key

$ openssl x509 -in cert.crt -text
Certificate:
```

```
  Data:
    Version: 1 (0x0)
    Serial Number:
      73:c5:2f:1b:06:2e:14:0c:7d:98:f0:f8:8f:c1:32:5f:b4:81:9b:4c
    Signature Algorithm: sha256WithRSAEncryption
    Issuer: CN = mitmproxy, O = mitmproxy
    Validity
      Not Before: Jan 22 23:02:59 2021 GMT
      Not After : Jan 22 23:02:59 2022 GMT
    Subject: C = ES, ST = Madrid, O = mitmproxy, CN = www.example.com
    Subject Public Key Info:
      Public Key Algorithm: rsaEncryption
        RSA Public-Key: (2048 bit)
        Modulus:
                              ...
        Exponent: 65537 (0x10001)
  Signature Algorithm: sha256WithRSAEncryption
    ...

$ cat cert.crt ca.crt cert.key > cert_key.pem

$ sudo mitmproxy --mode reverse:https://www.example.com:443 --listen-host 0.0.0.0 --listen-port 443 --cert www.example.com=cert_key.pem
```

Finalmente, para protocolos no basados en HTTP pero que utilicen TLS, se debe utilizar la opción '--tcp-hosts <*host*>' para que no interprete las conexiones TLS destinadas a ese *host* como flujos HTTP. La opción '--ignore-hosts <*host*>', deshabilita la interceptación TLS en las comunicaciones con ese *host*, y se comporta como un *proxy* convencional, reenviando el tráfico cifrado entre cliente y servidor.

En caso de necesitar analizar el tráfico a nivel de paquete, 'mitmproxy' también implementa la variable de entorno SSLKEYLOGFILE para indicar dónde debe almacenar las claves de las diferentes sesiones TLS que establece, y de esta forma utilizar 'Wireshark' para descifrar el tráfico (que se puede capturar en paralelo con 'tcpdump').

# Bibliografía adicional

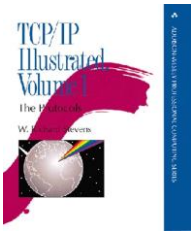



James Forshaw. "***Attacking Network Protocols***". No Starch Press. 2018.

Este libro cubre la mayoría del temario del curso, especialmente el capítulo sobre captura de tráfico, y el análisis de los diferentes tipos de aplicaciones TCP/IP.


Chris Sanders. "***Practical Packet Analysis***". No Starch Press. 2017.

Este libro se centra en el análisis de tráfico con 'Wireshark', con una gran cantidad de casos prácticos, y también repasa los principales protocolos TCP/IP.




W. Richard Stevens. "***TCP/IP Illustrated, Volume 1: The Protocols***". Primera edición. Addison Wesley Professional Computing, 1993.

Aunque ya está bastante desactualizado (e.g. solo habla de las direcciones basadas en clases en lugar de en máscaras de red) es un texto clásico sobre los protocolos de la torre TCP/IP


Ivan Ristic. "***Bulletproof SSL and TLS***". Feisty Duck. Junio 2017.

Este libro estudia en profundidad en el protocolo TLS/SSL.