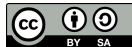
Redes de computadores

Prácticas de laboratorio

Juan Segarra Flor y Jesús Alastruey Benedé

Dpt. de informática e ingeniería de sistemas
Universidad de Zaragoza

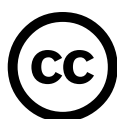
Curso 2023–2024



Este documento se realiza gracias al apoyo institucional de la Convocatoria competitiva de Proyectos de Innovación de la Universidad de Zaragoza (PI_DTOST) en el año 2023 y con referencia 4655 con título «Curso OCW para la asignatura Redes de computadores». Concretamente, el apoyo institucional ha consistido en una financiación de 0 €.

© 2011-2018 Juan Segarra (97,3 %) y Natalia Ayuso (2,7 %)
© 2019-2023 Juan Segarra (89,4 %) y Jesús Alastruey (10,6 %)

Esta obra está distribuida bajo una *Licencia Creative Commons BY-SA*. Para ver una copia de la licencia, visite `https://creativecommons.org/licenses/by-sa/4.0/legalcode.es`



Resumen de Licencia Creative Commons

Atribución-CompartirIgual 4.0 Internacional (CC BY-SA 4.0)

Este es un resumen legible por humanos (y no un sustituto) de la licencia.

Usted es libre de:

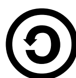
Compartir — copiar y redistribuir el material en cualquier medio o formato.

Adaptar — remezclar, transformar y construir a partir del material para cualquier propósito, incluso comercialmente.

El licenciante no puede revocar estas libertades en tanto usted siga los términos de la licencia

Bajo los siguientes términos:

Atribución — Usted debe dar crédito de manera adecuada, proporcionar un enlace a la licencia, e indicar si se han realizado cambios. Puede hacerlo de cualquier forma razonable, pero no de forma que sugiera que usted o su uso tienen el apoyo del licenciante.



CompartirIgual — Si remezcla, transforma o crea a partir del material, debe distribuir su contribución bajo la misma licencia del original.

No hay restricciones adicionales — No puede aplicar términos legales ni medidas tecnológicas que restrinjan legalmente a otras a hacer cualquier uso permitido por la licencia.

Avisos:

No tiene que cumplir con la licencia para elementos del material en el dominio público o cuando su uso esté permitido por una excepción o limitación aplicable. No se dan garantías. La licencia podría no darle todos los permisos que necesita para el uso que tenga previsto. Por ejemplo, otros derechos como publicidad, privacidad, o derechos morales pueden limitar la forma en que utilice el material.

Índice general

1. Análisis de tráfico y encapsulación de protocolos **5**
1.1. Objetivos 5
1.2. Entorno de trabajo 5
1.3. Captura de tráfico 5
1.4. Pila de protocolos 7
1.5. Direcciones de red 9
1.6. Identificadores de acceso al medio 9
1.7. Topología de red 10
1.7.1. Comunicación punto-a-punto y extremo-a-extremo 10
1.8. Puertos software 11
1.9. Capa de aplicación 11
1.10. ¿Sabías que...? 12

2. Interfaz *socket* y programación en red sobre TCP/IP **13**
2.1. Objetivos 13
2.2. Entorno de trabajo 13
2.3. Uso básico del manual del sistema (`man`) 13
2.4. Introducción a los *sockets* 14
2.4.1. Tipos de socket 14
2.4.2. Direcciones de red 14
2.4.3. Números de puerto 14
2.5. El modelo cliente-servidor 14
2.5.1. Cliente-servidor TCP 15
2.5.2. Implementación de un cliente TCP sencillo 16
2.5.3. Implementación de un servidor TCP sencillo 16
2.5.4. Ejecución de cliente y servidor TCP 18
2.6. Estructuras de direcciones 19
2.6.1. Implementación de *migetaddrinfo* 20
2.6.2. Preguntas de comprensión 21
2.7. Implementación de programas cliente/servidor 21
2.8. Evaluación de la práctica 22
2.9. ¿Sabías que...? 22
2.A. Código *migetaddrinfo* (`migetaddrinfo.c`) 22
2.B. Código común para gestionar direcciones (`comun.c`) 24
2.C. Código *cliente* de contar vocales (`clientevocalesTCP.c`) 27
2.D. Código *servidor* de contar vocales (`servidorvocalesTCP.c`) 31

3. Programación en red sobre UDP **37**
3.1. Objetivos 37
3.2. Programación usando UDP 37
3.2.1. Detalles de implementación 37
3.2.2. Servidor para pruebas 38

3.2.3. Preguntas de comprensión . . . 38
3.3. Automatización de la compilación con `make` . . . 39
3.4. Capa de presentación . . . 39
3.5. Evaluación de la práctica . . . 39
3.6. ¿Sabías que...? . . . 39
3.A. `Makefile` para fuentes de contar vocales UDP . . . 40

4. Implementación de protocolos y algoritmos de secuenciación 41
4.1. Objetivos . . . 41
4.2. Trabajo a realizar . . . 41
4.3. Protocolo RCFTP . . . 42
4.3.1. Formato del mensaje . . . 42
4.3.2. Flags . . . 43
4.4. Servidor RCFTPd . . . 43
4.4.1. Traza del servidor . . . 44
4.5. Algoritmo básico . . . 47
4.5.1. Ayuda a la implementación del algoritmo básico . . . 47
4.5.2. Ayuda para probar el algoritmo básico . . . 49
4.5.3. Preguntas sobre el funcionamiento . . . 49
4.6. Algoritmo *Stop&Wait* . . . 49
4.6.1. Gestión de timeouts mediante *multialarm* . . . 50
4.6.2. Interrupciones de llamadas al sistema . . . 50
4.6.3. Ayuda a la implementación del algoritmo *Stop&Wait* . . . 50
4.6.4. Preguntas sobre el funcionamiento . . . 51
4.7. Algoritmo *Go-Back-n* . . . 51
4.7.1. Gestión de timeouts e interrupciones . . . 52
4.7.2. Ayuda a la implementación del algoritmo *Go-Back-n* . . . 52
4.7.3. Preguntas sobre el funcionamiento . . . 53
4.8. Evaluación del trabajo . . . 53
4.9. ¿Sabías que...? . . . 54

5. Topología y tráfico en Internet 55
5.1. Objetivos . . . 55
5.2. Historia . . . 55
5.3. Topología y tráfico en Internet . . . 55
5.3.1. Tráfico entre SAs . . . 57
5.3.2. Consultas mediante WHOIS . . . 58
5.4. Herramienta `ping` . . . 59
5.5. Herramienta `traceroute` . . . 59
5.6. ¿Sabías que...? . . . 60

6. Herramientas básicas de red 61
6.1. Objetivos . . . 61
6.2. Introducción . . . 61
6.3. Interfaces de red . . . 61
6.4. Conectividad local . . . 62
6.5. Tablas de reexpedición/encaminamiento . . . 62
6.5.1. IPv4 . . . 62
6.5.2. IPv6 . . . 63
6.6. Servidores de nombres de dominio . . . 64
6.7. Estado de puertos . . . 65
6.8. Interacción con protocolos de aplicación . . . 66
6.9. Herramienta Nmap . . . 66
6.10. ¿Sabías que...? . . . 67

Práctica 1

Análisis de tráfico y encapsulación de protocolos

1.1. Objetivos

Aprendizaje básico de la arquitectura de red y la encapsulación de protocolos usando el analizador de tráfico *Wireshark*. Análizar la topología básica de una red. Comprender las diferencias entre conexión punto-a-punto y extremo-a-extremo.

1.2. Entorno de trabajo

Las prácticas de la asignatura se realizarán en el sistema *CentOS* (GNU/Linux) de los equipos del laboratorio L1.02, al que se accede con el nombre de usuario y contraseña de *Hendrix*. En cualquier momento puedes cambiar la contraseña desde `https://diis.unizar.es/WebEstudiantes/`.

1.3. Captura de tráfico

Existen muchos programas para capturar el tráfico que circula por una red. Por ejemplo, TShark puede ejecutarse desde un terminal y permite visualizar dicho tráfico, tal y como se muestra en el siguiente ejemplo:

```
lab102-208:~/ tshark -i enp3s0
Capturing on 'enp3s0'
    1 0.000000000 85.251.19.126 ? 155.210.154.208 TCP 66 58608 ? 22 [ACK] Seq=1 Ack=1 Win=501
      Len=0 TSval=897855530 TSecr=959665737
    2 0.176475971 Cisco_f3:c7:c3 ? Broadcast ARP 60 Who has 155.210.154.222? Tell 155.210.154.254
    3 0.381764319 Cisco_f3:c7:c3 ? Broadcast ARP 60 Who has 155.210.154.51? Tell 155.210.154.254
    4 0.391699371 Cisco_f3:c7:c3 ? Broadcast ARP 60 Who has 155.210.154.31? Tell 155.210.154.254
    5 0.488590113 155.210.154.208 ? 85.251.19.126 SSH 438 Server: Encrypted packet (len=372)
    6 0.516116734 85.251.19.126 ? 155.210.154.208 TCP 66 58608 ? 22 [ACK] Seq=1 Ack=373 Win=501
      Len=0 TSval=897856049 TSecr=959666256
    7 0.703471668 Cisco_f3:c7:c3 ? Broadcast ARP 60 Who has 155.210.154.142? Tell 155.210.154.254
    8 0.995384843 155.210.154.208 ? 85.251.19.126 SSH 302 Server: Encrypted packet (len=236)
    9 0.996171266 155.210.154.208 ? 85.251.19.126 SSH 342 Server: Encrypted packet (len=276)
   10 1.021863290 85.251.19.126 ? 155.210.154.208 TCP 66 58608 ? 22 [ACK] Seq=1 Ack=609 Win=501
      Len=0 TSval=897856556 TSecr=959666763
10 packets captured
```

En esta práctica vamos a utilizar *Wireshark*, un analizador de protocolos de red con entorno gráfico. Se puede lanzar desde el menú de aplicaciones. Al iniciar el programa, dependiendo de la versión instalada, aparece una ventana similar a la mostrada en la figura 1.1. En la parte central izquierda se puede seleccionar de un listado el interfaz de red por el que capturar el tráfico. El interfaz de captura también

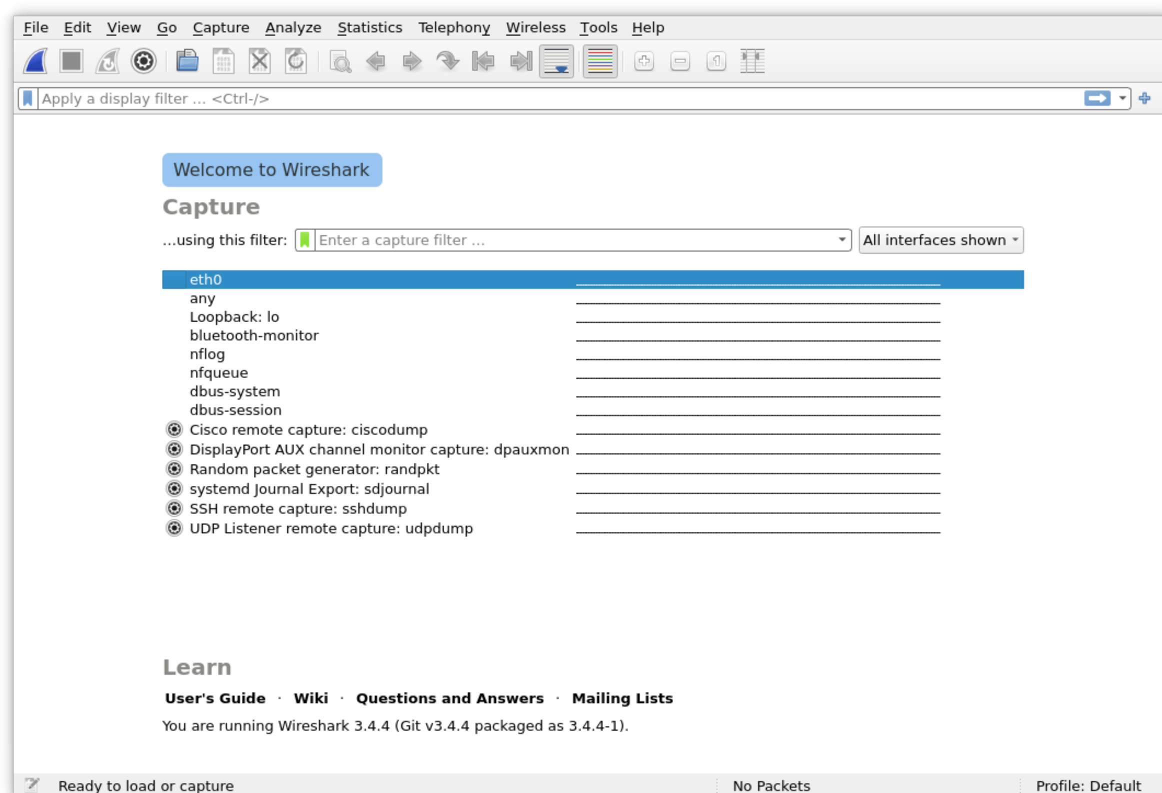


Figura 1.1: Pantalla de inicio del programa *Wireshark*

puede elegirse desde las opciones del menú de captura. En esta práctica hay que capturar el tráfico cursado por el interfaz de red asociado a la dirección IP pública del equipo, que debería ser «br0». Para comprobarlo, ejecuta en un terminal `ifconfig -a`. Por ejemplo, en el caso de la máquina lab102-208, cuya dirección IP es la 155.210.154.208, el interfaz a seleccionar sería «br0»:

```
lab102-208:~/ ifconfig -a
br0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST> mtu 1500
       inet 155.210.154.208 netmask 255.255.255.0 broadcast 155.210.154.255
       inet6 fe80::a31:7036:fb8a:fdda prefixlen 64 scopeid 0x20<link>
       inet6 2001:470:1f0b:19fb:c084:fee7:f3b4:ed4e prefixlen 64 scopeid 0x0<global>
       ether 00:10:18:80:67:84 txqueuelen 1000 (Ethernet)
       RX packets 29755 bytes 3438627 (3.2 MiB)
       RX errors 0 dropped 0 overruns 0 frame 0
       TX packets 4003 bytes 560926 (547.7 KiB)
       TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0
[...]
```

El analizador captura las tramas (*frames*) que circulan por el medio de transmisión, que en este caso es el cable de par trenzado, e interpreta su formato, en este caso Ethernet. Cada trama puede incluir distintos protocolos, y el analizador visualiza el contenido de los campos de sus cabeceras. Para ello, la pantalla del programa se divide en varias áreas de visualización (figura 1.2):

1. Área de definición de filtros. Justo bajo los botones se pueden especificar filtros, de forma que el resto de las áreas sólo muestren entradas que coincidan con el filtro. Por ejemplo puedes escribir «tcp» para que solo muestre los paquetes TCP capturados.

2. Área de visualización de tramas. En ella aparece el listado de tramas capturadas, con su información

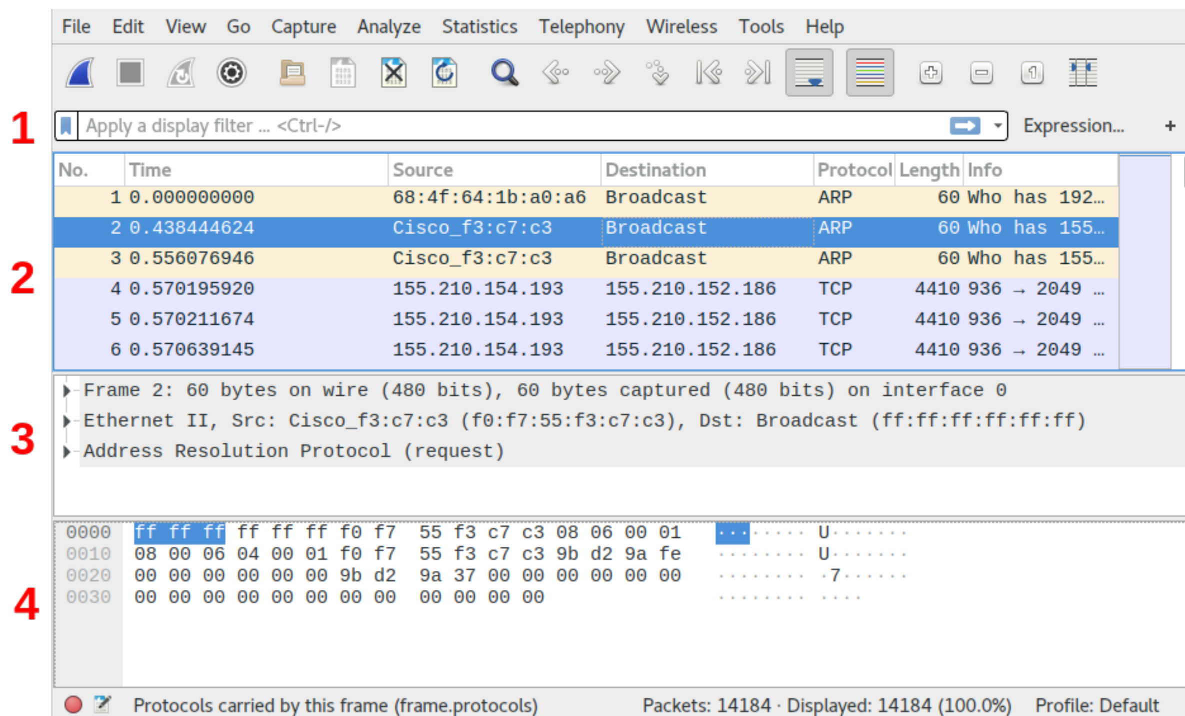


Figura 1.2: Áreas de visualización del programa *Wireshark*

básica: número de trama capturada, instante de captura, direcciones origen y destino, protocolo de la capa más alta en la trama, e información abreviada de su contenido.

3. Área de detalle de una trama. Seleccionando una de las tramas del área de visualización se muestra toda su información mediante detalles *desplegables*. Los detalles incluyen la información de los campos de cada protocolo, que se estudiarán más adelante en la asignatura.

4. Área de información en bruto. En la parte inferior se muestra la trama (y texto para caracteres imprimibles), es decir, los unos y ceros que realmente forman la trama. Por ejemplo, los datos de aplicación transmitidos se pueden visualizar en esta parte.

Además, en la zona superior se encuentran los menús y botones, desde los que se puede iniciar/parar una captura, guardarla, etc.

1.4. Pila de protocolos

En clase de teoría hemos visto las distintas capas en la arquitectura de red, que se implementan mediante distintos protocolos que proporcionan servicios específicos. Inicia una captura de tráfico mientras accedes con un navegador a algún sitio web, por ejemplo, `http://diis.unizar.es`. Al seleccionar una trama, en la parte desplegable se pueden ver sus detalles. La lista ordenada de protocolos siguiendo la estructura de capas (*pila de protocolos*) que intervienen en la trama seleccionada se puede ver en la parte desplegable de trama, en la línea `[Protocols in frame: ]`. Aunque suele haber un protocolo por capa, ten en cuenta que no todas las comunicaciones necesitan usar todas las capas. También es posible que un mismo protocolo tenga variantes. Por ejemplo el protocolo Ethernet habitual tiene un campo *ethertype* que no estaba en el protocolo original. Dependiendo de la versión del Wireshark, puede aparecer la palabra *ethertype* en `[Protocols in frame: ]` para proporcionar este detalle, aunque no sea un protocolo.

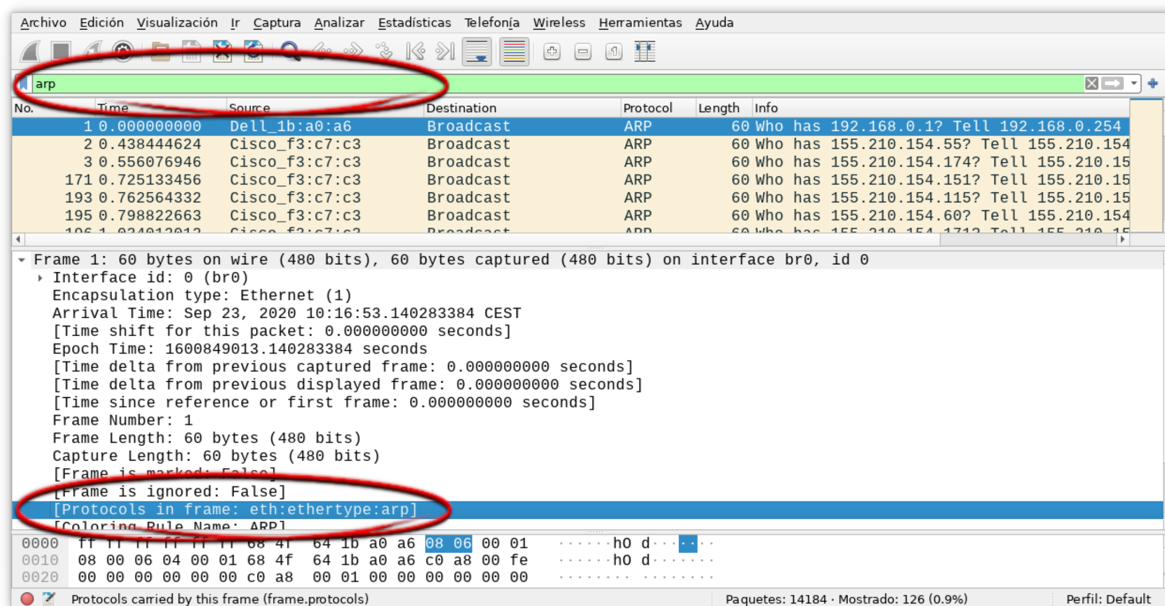


Figura 1.3: Uso de filtro y visualización de pila de protocolos (`[Protocols in frame: ]`)

Filtra tramas etiquetadas con los siguientes protocolos y anota qué pila de protocolos usan dichas tramas (ejemplo en figura 1.3).

1. ARP (*Address Resolution Protocol*). Protocolo de apoyo a la capa de red para preguntar qué máquina tiene cierto identificador ethernet/MAC:

2. DNS (*Domain Name Service*). Protocolo de la capa de aplicación para preguntar por nombres de equipos:

3. HTTP (*HyperText Transfer Protocol*). Protocolo de la capa de aplicación de comunicación web (necesitarás capturar una web sin cifrado, por ejemplo `http://diis.unizar.es/`):

4. TCP (*Transmission Control Protocol*). Protocolo fiable de la capa de transporte:

Observa el orden en el que se muestran los protocolos de una trama (pila de protocolos).

5. Teniendo en cuenta el encapsulado en una trama con TCP y HTTP, ¿la parte TCP está dentro de los datos de HTTP, o es la parte HTTP la que está dentro de los datos de TCP?

6. Dibuja el esquema de encapsulado (zona de cabeceras y datos de protocolos para cada capa, como en la transparencia de *encapsulación* vista en clase, sin indicar los bytes) de una trama que contenga el protocolo HTTP.

Como habrás podido observar en la captura, existen muchos protocolos y no siempre se usa un protocolo concreto en una capa concreta de la arquitectura. Dispones de un listado con los protocolos de *sistema* más relevantes en el fichero `/etc/protocols`. Puedes verlo por ejemplo desde el navegador accediendo a `file:///etc/protocols/` o desde el explorador de archivos del sistema. Observa que no aparecen los protocolos de la capa de aplicación (e.g. HTTP), ya que éstos se implementan en las aplicaciones y no en el sistema.

7. ¿Cuál es la descripción que aparece asociada al protocolo UDP en ese fichero?

8. ¿Cuál es la descripción que aparece asociada al protocolo ICMP en ese fichero?

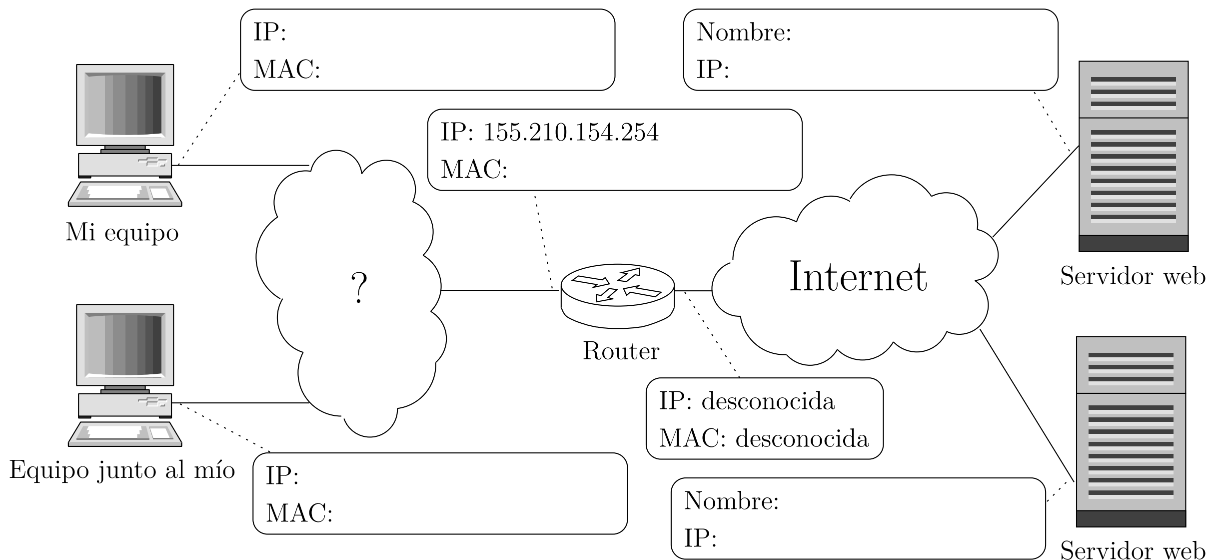


Figura 1.4: Esquema del laboratorio y de dos servidores web en Internet

1.5. Direcciones de red

La red Internet está construida alrededor del protocolo IP (*Internet Protocol*), situado en la capa de red. La cabecera de este protocolo incluye campos donde se encuentran codificadas las direcciones de los equipos en Internet. Las direcciones IPv4 se muestran como cuatro valores entre 0 y 255 separados por puntos, e.g. 155.210.152.177. Las direcciones IPv6 se muestran como ocho bloques entre 0 y 0xffff separados por dos puntos (bloques de 0s consecutivos se pueden omitir), e.g. 2001:720:418:cafd::20. Accede a varios sitios de Internet y responde a las siguientes preguntas.

9. Comparando la información de varias tramas, ¿cuál puedes deducir que es la dirección IP de tu equipo? ¿Coincide con la de la pegatina identificativa que lleva tu equipo? Anota la IP en la parte correspondiente a «Mi equipo» en la figura 1.4. Anota también la IP de dos de los servidores Web consultados.

10. Pregunta a tus compañeros qué dirección IP tienen sus equipos. ¿Se parecen a la de tu equipo? ¿En qué se parece? Anota la IP de alguno de los equipos dispuestos en tu misma bancada en la parte correspondiente a «Equipo junto al mío» en la figura 1.4.

11. Con la información anterior, ¿se podría deducir que equipos «cercanos» deben tener direcciones IP «cercanas»?

1.6. Identificadores de acceso al medio

Además de las direcciones «lógicas» anteriores, cada tarjeta de red tiene un identificador «físico» (también llamado dirección MAC, *media access control*, o dirección *ethernet*). Este identificador se codifica en campos del protocolo Ethernet o del protocolo de acceso al medio que corresponda. Para Ethernet y WiFi, el identificador MAC se muestra como seis valores entre 0 y 0xff separados por dos puntos, e.g. 00:10:18:80:6e:1f. El identificador MAC está asociado a la tarjeta de red, es único, viene establecido por el fabricante, y no es posible modificarlo.

12. Al mostrar los identificadores MAC, en algunos casos la parte de mayor peso del identificador se muestra como un nombre. ¿Qué puede indicar ese nombre? ¿Cómo puede saber ese nombre el analizador?

13. Conociendo la dirección IP de tu equipo y viendo el detalle de las tramas, ¿cuál es el identificador ethernet/MAC de tu equipo? ¿Coincide con la de la pegatina identificativa que lleva tu equipo? Anótalo en la etiqueta correspondiente de la figura 1.4.
14. Busca varias tramas ARP y observa los identificadores MAC *destino*. ¿Hay alguno que resulte especialmente curioso? Teniendo en cuenta los tipos de envío, por destino, ¿a qué crees que corresponde dicho identificador?
15. Haz una captura mientras accedes a distintos lugares de Internet y anota los identificadores MAC de las tramas que recibes de ellos. ¿Se parecen? ¿Por qué? ¿A qué equipo corresponde el identificador MAC que no es el de tu propio equipo? Revisa la transparencia «Retransmisores» vista en teoría si tienes dificultades para responder estas cuestiones.
16. Compara el identificador MAC de tu equipo con el de la pregunta anterior. ¿Se parecen?
17. Con la información anterior, ¿se podría deducir que los identificadores MAC dependen del fabricante y no de la «cercanía» de los equipos?

1.7. Topología de red

La topología física y lógica del laboratorio de prácticas no es sencilla. Sin embargo, vamos a determinar si los equipos del laboratorio configuran una red en bus o estrella observando el tráfico capturado con *Wireshark*. Para ello:

18. Teniendo en cuanta que *Wireshark* captura *todo* lo que pasa por el cable de red, ¿deberías poder capturar desde tu equipo el tráfico generado y recibido por los equipos vecinos si hay una topología en bus? ¿Y si la topología es en estrella?
19. Realiza la prueba anterior para verificar la topología del laboratorio. ¿Es una topología en bus o en estrella?

Recuerda que existen dispositivos de interconexión *transparentes*, es decir, que no modifican las tramas que los atraviesan, y por lo tanto no dejan pistas para poder descubrirlos simplemente observando el tráfico.

20. ¿Es transparente el encaminador (*router*)?
21. Desde un terminal, introduce el comando `ping <destino>` sustituyendo `<destino>` por la dirección IP de cualquier equipo del laboratorio que esté encendido. Busca las tramas correspondientes a ese tráfico y anota los identificadores MAC de la captura. ¿Coinciden con los del encaminador? ¿Han pasado esas tramas por el encaminador?
22. Teniendo en cuenta la topología deducida anteriormente, y que no todo el tráfico pasa por el encaminador, ¿qué podemos deducir que hay entre los equipos y el encaminador? Dibuja la topología en la nube etiquetada con «?» de la figura 1.4.

1.7.1. Comunicación punto-a-punto y extremo-a-extremo

Responde a las siguientes preguntas *sin tener en cuenta la posible presencia de equipos transparentes*:

23. Si te conectas con un servidor web de google, ¿es una comunicación punto-a-punto, extremo-a-extremo o ambas?
24. Si te conectas con un servidor web de google, ¿la comunicación *entre tu equipo y el encaminador* es punto-a-punto, extremo-a-extremo o ambas?
25. Si haces `ping` al equipo junto al tuyo, ¿es una comunicación punto-a-punto, extremo-a-extremo o ambas?
26. ¿Cuál es la diferencia entre una comunicación punto-a-punto y otra extremo-a-extremo?

1.8. Puertos software

Cada una de las tramas que llega a un equipo va dirigida a un proceso específico en ejecución en el equipo. Igual que hay interfaces (puertos/conectores) hardware, la arquitectura de red TCP/IP define puertos software que asocian una transmisión (capa de transporte) y un proceso. Estos puertos software podrían considerarse los identificadores de la capa de transporte. Para que un cliente, por ejemplo un navegado web, sea atendido por cierto proceso servidor en otro equipo, por ejemplo un servidor web, debe usar un protocolo de transporte determinado y el puerto software que tiene asociado. Aunque el proceso servidor se puede asociar a cualquier protocolo/puerto, cada servicio suele tener un protocolo/puerto «recomendado», especificado en el fichero `/etc/services`. De esta forma, el proceso cliente sabe a qué protocolo y puerto software dirigir su transmisión, dependiendo del servicio deseado.

27. ¿Qué puertos usan los servicios *http*, *imap3* y *ssh* según el fichero anterior?

28. Realiza varias peticiones web desde un navegador con múltiples pestañas y observa con el *wireshark* los puertos implicados. ¿A qué puerto enviamos las peticiones web? ¿Siempre a ese puerto?

29. ¿Desde qué puerto nos responden? ¿Coincide con el anterior?

30. ¿Desde qué puerto enviamos las peticiones? ¿Siempre desde ese puerto?

31. ¿A qué puerto nos responden? ¿Coincide con el anterior?

1.9. Capa de aplicación

Para esta parte, recuerda que puedes utilizar filtros. Por ejemplo, el filtro «http» visualiza sólo las tramas que contengan el protocolo HTTP, y el filtro «http.request» visualiza sólo las tramas que contengan peticiones HTTP.

32. Accede a `http://diis.unizar.es` y localiza las tramas de respuesta HTTP. La parte inferior del analizador muestra la información transmitida en hexadecimal y en texto. ¿Qué información se ve?

33. Accede a `https://moodle.unizar.es` y observa que el navegador indica mediante algún símbolo que está utilizando cifrado. Introduce tu usuario/contraseña y captura el tráfico que se genera al hacer clic en *Log in*. Puedes localizar las tramas correspondientes filtrando por número de puerto 443 en TCP o por protocolo (dependiendo de la versión de Wireshark, podría mostrarse como protocolo SSL, TLS o HTTPS). ¿Puedes ver el usuario/contraseña?

34. ¿Se verá el usuario/contraseña de servicios (web, correo, mensajería, etc.) que no vayan cifrados?

35. Aunque el analizador está configurado para que funcione sin ser administrador en el laboratorio, ¿tiene sentido que necesite permisos de administrador en un sistema multiusuario?

36. El analizador captura el tráfico que «pasa» por el medio de transmisión al que estamos conectados. Asumiendo que el medio de transmisión (cable de red) de nuestro equipo *no se comparte* con otros usuarios, ¿qué otros equipos verían el tráfico que hemos generado para acceder a Google?

37. Si estamos usando un medio compartido, como por ejemplo el aire en una conexión inalámbrica, ¿qué otros equipos verían el tráfico además de los anteriores?

Vamos a crear ahora nuestra propia aplicación mediante el programa *netcat*: un *chat* entre dos personas. Elige un compañero que esté en otro equipo, con quien entablar la comunicación. Abrid en ambos equipos un terminal. En uno de los equipos ejecutad el comando `nc -l -p 32005` para *escuchar* por el puerto 32005. En el otro equipo ejecutad el comando `nc <destino> 32005` donde habrá que sustituir `<destino>` por la dirección IP del otro equipo, que habréis obtenido en una de las preguntas anteriores.

38. Una vez esté establecida la comunicación, escribe algo en cualquiera de los dos lados. ¿Qué sucede?

39. Usa el analizador para capturar especificando el filtro «tcp.port==32005» para mostrar solamente mensajes que usen el puerto 32005. ¿Aparecen los mensajes que estas generando/recibiendo? ¿Cuál es la pila de protocolos que se está usando?

De forma similar, podemos utilizar esta herramienta para interactuar con protocolos de aplicación basados en mensajes de texto. Activa la captura de tráfico, esta vez filtrando el puerto 80.

40. Lanza `nc -C webdiis.unizar.es 80` y escribe exactamente, respetando mayúsculas y minúsculas, y sin olvidar la línea en blanco final:

```
GET / HTTP/1.1
Host: webdiis.unizar.es

```

¿Qué ha respondido webdiis a tu mensaje? ¿Puedes verlo en el analizador?

En las secciones anteriores hemos visto identificadores usados en protocolos concretos, situados en distintas capas. Para protocolos en la capa de aplicación también pueden existir identificadores, aunque es menos común. Dado que con los identificadores lógicos anteriores (dirección IP y puerto software) ya estamos especificando un servicio, sólo sería necesario un identificador en capa de aplicación si dentro de dicho servicio estuviéramos realmente ofreciendo varios servicios.

41. Usando el navegador, accede a `http://www.fechadehoy.com/`. ¿Cuál es su dirección IP y número de puerto?

42. Accede ahora a `http://www.convertidorunidades.com/`. ¿Cuál es su dirección IP y número de puerto?

43. ¿Qué muestra el navegador al acceder a `http://50.116.5.124/`?

44. ¿Qué puede usar como identificador de aplicación el protocolo HTTP?

1.10. ¿Sabías que...?

- En `https://diis.unizar.es/es/estudiantes` dispones de recursos útiles para estudiantes. Por ejemplo, desde ahí puedes cambiar la contraseña de acceso a los equipos de prácticas, o informarte sobre cómo limpiar el espacio en disco si superas tu *quota* asignada.

- *Wireshark* es un analizador bajo licencia pública general (GPL) (código fuente libre), que puedes descargar, modificar, copiar, etc. La mayoría de distribuciones GNU/Linux ofrecen el *wireshark* en forma de paquete, con lo que no es necesario descargarlo manualmente.

- Un alumno de informática la Universidad de Zaragoza fue acusado de suplantar la página web de conexión inalámbrica Wiuz, en la que se pide usuario y contraseña. Seguro que haría bien esta práctica, pero... *un gran poder conlleva una gran responsabilidad*.

Práctica 2

Interfaz *socket* y programación en red sobre TCP/IP

2.1. Objetivos

Uso básico del manual del sistema. Introducción al modelo cliente-servidor, abstracción *socket* y estructuras de direcciones. Programación de una aplicación sencilla TCP.

2.2. Entorno de trabajo

Se recomienda realizar esta práctica en los equipos locales del laboratorio con GNU/Linux. No se recomienda realizar esta práctica en `hendrix.cps.unizar.es`. Aparte de los equipos del laboratorio, dispones del equipo `lab000.cps.unizar.es`, al que se puede acceder de forma remota mediante `ssh`, que es equivalente a los equipos de los laboratorios. Puedes usarlo para completar las prácticas fuera de horarios o para realizarlas de forma no presencial.

2.3. Uso básico del manual del sistema (`man`)

Todos los sistemas *NIX tienen su documentación detallada en las páginas del manual, que se pueden visualizar mediante el comando `man`. Si no conoces las convenciones de sintaxis para describir parámetros de comandos puedes consultar la sección *Utility Argument Syntax*[1]. Para visualizar una página concreta, hay que usar como parámetro el nombre de la página deseada: man <página>. Por ejemplo, el comando `man man` mostrará la página del comando `man`.

Cuando el sistema muestra la página del manual, normalmente lo hace a través del paginador `less`. La mayoría de los sistemas responden a las teclas [↑], [↓], [Av Pág], [Re Pág] y [   ], pero es muy útil conocer algunos comandos más, por ejemplo:

- /<texto> busca <texto> (resalta en pantalla todas las apariciones y va directamente a la siguiente aparición). Si no se pone <texto> repite la búsqueda anterior.
- `?` funciona igual que `/` pero busca la aparición previa.
- <núm>G va directamente a la línea <núm>. Para ir al final, donde a veces hay ejemplos y funciones relacionadas, `0G`.
- `q` sale del paginador (y de la página del manual).

1. Visualiza la página del manual para la llamada `socket()`. ¿Cuál es el comando para hacerlo?

[1] http://pubs.opengroup.org/onlinepubs/9699919799/basedefs/V1_chap12.html

El manual está dividido en *secciones* (código entre paréntesis que aparece en la primera línea al lado de la página del manual que se esté consultando). Aunque no es muy frecuente, puede haber páginas con el mismo nombre en distintas secciones. Por ejemplo, *shutdown* es un comando del sistema y una llamada al sistema. Para listar las páginas (y su sección) cuya descripción contenga una palabra clave se puede usar `man -k <palabra_clave>`, y para especificar una sección concreta `man -s <sección> <página>`.

2. Consulta el manual para `shutdown` en la sección 8 (herramientas de administración). ¿Qué hace el comando `shutdown`?

3. Consulta el manual para *shutdown* en la sección 2 (llamadas al sistema). ¿Qué hace la llamada `shutdown()`?

2.4. Introducción a los *sockets*

La comunicación en red se basa en una abstracción llamada *socket*. Un socket (enchufe) representa uno de los extremos de una conexión bidireccional entre dos procesos. Cuando un proceso lo necesita, pide al sistema operativo la creación de un socket. El sistema devuelve un «descriptor» que el proceso usará para referirse al nuevo socket. Dependiendo de las características deseadas para la comunicación (en general TCP o UDP) se solicitará el tipo de socket correspondiente. Los dos extremos de la comunicación deben tener *el mismo tipo de socket*. Para poder crear un socket, hay que especificar tipo de comunicación, direcciones y número de puerto locales.

2.4.1. Tipos de socket

En la pila de protocolos TCP/IP en general se usan dos tipos de socket:

SOCK_STREAM: proporciona una transmisión bidireccional continua y fiable (los datos se reciben ordenados, sin errores, sin pérdidas y sin duplicados) de bytes *con conexión* mediante el protocolo TCP (*Transmission Control Protocol*).

SOCK_DGRAM: proporciona una transmisión bidireccional no fiable, de longitud máxima prefijada, *sin conexión* mediante el protocolo UDP (*User Datagram Protocol*).

2.4.2. Direcciones de red

Una dirección de red es un identificador lógico de un interfaz de red que permite su localización de forma jerárquica. En la familia de protocolos de Internet las direcciones IPv4 son de 32 bits, mientras que las IPv6 son de 128 bits.

2.4.3. Números de puerto

Los puertos asocian sockets a procesos y se identifican por un número entero sin signo de 16 bits (rango de 0 a 65535). Los puertos 0 a 1023 están reservados para los servicios «bien conocidos» (*well-known ports*) y requieren privilegios para su uso. Por ejemplo, el puerto 80 está reservado para el servicio web (protocolo HTTP). En la práctica anterior ya viste que puedes consultar el puerto que ocupa cada servicio en el archivo `/etc/services`.

2.5. El modelo cliente-servidor

El modelo más utilizado para el desarrollo de aplicaciones en red es el cliente-servidor:

a) El proceso servidor es un programa en ejecución que está a la espera de que algún cliente requiera sus servicios.

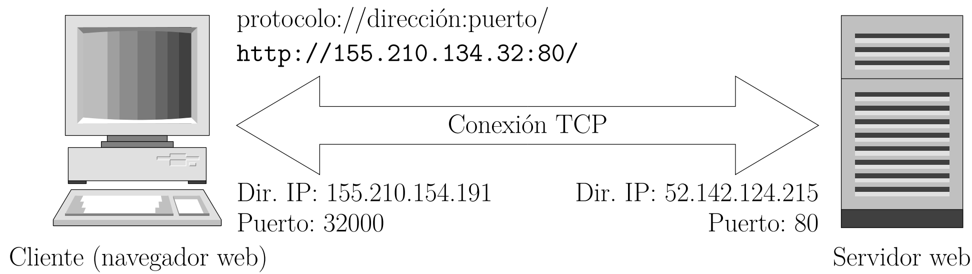


Figura 2.1: Conexión de un cliente a un servidor web

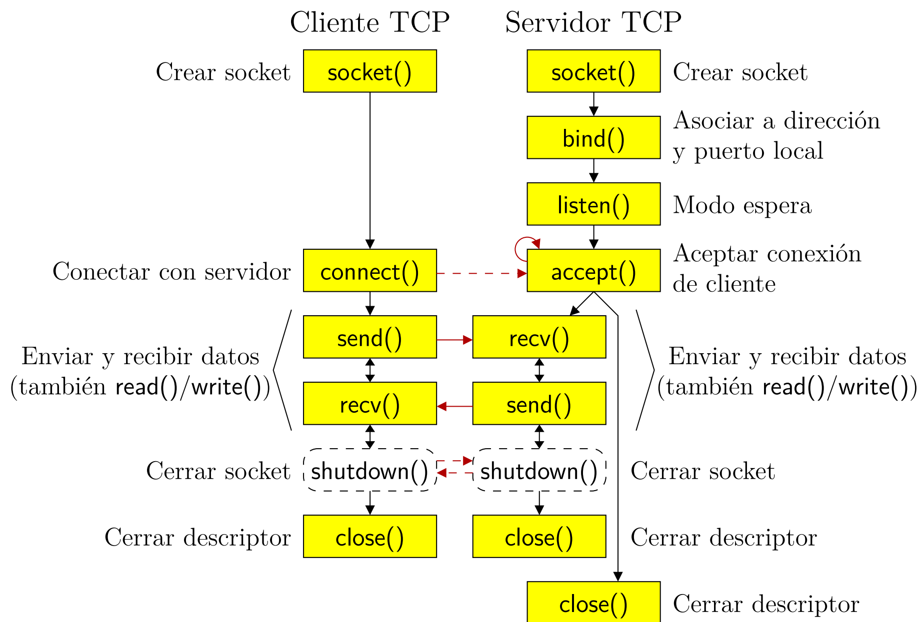


Figura 2.2: Llamadas al sistema para sockets en un protocolo orientado a conexión

b) Un proceso cliente, en ejecución en el mismo o en otro computador de la red, envía una petición de servicio hacia el servidor.

c) El servidor recibe la petición, responde al cliente y queda de nuevo a la espera de nuevas peticiones.

Un ejemplo de aplicación que sigue este modelo es el de cliente-servidor web. El servidor web espera que un cliente (navegador) le solicite una página web, como puede verse en la figura 2.1.

Para que la comunicación entre cliente y servidor sea posible, es necesaria la participación de una serie de protocolos de red. En nuestro caso nos vamos a centrar en la pila de protocolos TCP/IP. Cliente y servidor son normalmente procesos de usuario, mientras que los protocolos de transporte y red están implementados en el núcleo del sistema operativo. Los protocolos TCP y UDP se sitúan en la capa de transporte. Esta capa se encarga de comunicar dos procesos concretos, cada uno en un extremo de la comunicación, identificados mediante un número de puerto. Dependiendo del tipo de transmisión deseada, algunas aplicaciones usan TCP, UDP o ambos. En esta práctica trabajaremos con TCP.

2.5.1. Cliente-servidor TCP

Para una comunicación *con* conexión (TCP), hay que seguir una serie de pasos, tal y como se muestra en la figura 2.2.

El servidor creará inicialmente un extremo de la conexión pidiendo un socket —socket()— y asociándolo a una dirección local —bind()—. En TCP/IP una dirección local es la dirección IP de la máquina más un número de puerto sobre un protocolo de transporte (TCP o UDP). El servidor puede en algún momento recibir varias peticiones de conexión simultáneas por lo que se debe especificar el número máximo de conexiones en espera —listen()—. A continuación, si hay alguna conexión pendiente la atenderá —accept()—, en caso contrario, se quedará bloqueado a la espera de alguna conexión entrante.

Por su parte, el cliente también debe crear un socket. *Se desaconseja* el uso de bind() en el cliente, puesto que el cliente puede usar cualquier puerto libre y no es necesario especificar uno concreto. Una vez creado el socket, lanzará una petición de conexión al servidor —connect()—. Si el servidor está disponible, es decir, si ha ejecutado accept() y no hay peticiones anteriores en cola, inmediatamente se desbloquean tanto cliente como servidor. En la parte del servidor, accept() habrá devuelto un nuevo identificador del socket que está conectado con el cliente. El identificador del socket original sigue sirviendo para atender nuevas peticiones de conexión a medida que se vayan realizando llamadas accept(). Cliente y servidor se intercambiarán datos mediante send() y recv(). Opcionalmente, se puede cerrar uno o ambos sentidos de la conexión con shutdown(). Finalmente hay que cerrar el descriptor de socket mediante la llamada close(), que puede forzar el cierre de la conexión en caso de que siga abierta. En los sistemas Unix/Linux también es posible usar las llamadas read() y write() para leer y escribir en un socket.

2.5.2. Implementación de un cliente TCP sencillo

Vamos a considerar un ejemplo para ilustrar algunos de los conceptos que se han presentado. La figura 2.3 muestra la implementación de un cliente TCP. Este cliente establece una conexión con el puerto especificado de un sistema identificado por su dirección IP y muestra por salida estándar la cadena de texto enviada por el servidor. A continuación se describen las acciones principales relacionadas con la interfaz socket.

Crear un socket TCP. En la línea 12, la función socket() crea un socket de la familia Internet AF_INET de tipo SOCK_STREAM, es decir, un socket TCP. Esta función devuelve un número entero que se utilizará para identificar al socket en futuras llamadas a funciones.

Especificar dirección IP y puerto del servidor. En las líneas 17-19 se rellena la variable servaddr, que es una estructura de direcciones IPv4 (tipo struct sockaddr_in). En concreto, se especifica familia Internet AF_INET, y la dirección IP y puerto que se han pasado como parámetros por línea de comandos (argv[1] y argv[2]). La dirección IP y puerto en esta estructura deben almacenarse en un formato establecido, por eso se llama a las funciones inet_pton() (*presentation to numeric*) y htons() (*host to network short*) para realizar las conversiones apropiadas.

Establecer conexión con el servidor. En la línea 24, la función connect() establece una conexión TCP con el proceso del servidor especificado como parámetro (servaddr).

Leer mensaje del servidor. En la línea 29, la función read() recibe el mensaje del servidor.

Terminar el proceso. La función exit() en la línea 35 finaliza la ejecución del proceso. Unix/Linux cierra los descriptores de fichero abiertos de un proceso que termina, por lo que el socket se cierra.

2.5.3. Implementación de un servidor TCP sencillo

La figura 2.4 muestra la implementación de un servidor TCP. Este servidor recibe una petición de conexión, la acepta y envía al cliente una cadena de texto con la fecha y hora. A continuación se describen las acciones principales relacionadas con la interfaz socket.

Crear un socket TCP. El proceso de creación del socket en la línea 13 es idéntico al realizado por el cliente.

```
1  int main(int argc, char **argv)
2  {
3      int sockfd, n;
4      char recvline[MAXLINE + 1];
5      struct sockaddr_in  servaddr = { 0 };
6
7      if (argc != 3)
8      {
9          printf("usage: %s <IPaddress> <puerto>\n", argv[0]);
10         exit(0);
11     }
12     if ( (sockfd = socket(AF_INET, SOCK_STREAM, 0)) < 0)
13     {
14         printf("socket error\n");
15         exit(0);
16     }
17     servaddr.sin_family = AF_INET;
18     servaddr.sin_port   = htons(atoi(argv[2]));
19     if (inet_pton(AF_INET, argv[1], &servaddr.sin_addr) <= 0)
20     {
21         printf("inet_pton error for %s", argv[1]);
22         exit(0);
23     }
24     if (connect(sockfd, (struct sockaddr *) &servaddr, sizeof(servaddr)) < 0)
25     {
26         printf("connect error");
27         exit(0);
28     }
29     n = read(sockfd, recvline, MAXLINE);
30     if (n > 0)
31     {
32         recvline[n] = 0;     /* null terminate */
33         printf("%s\n", recvline);
34     }
35     exit(0);
36 }
```

Figura 2.3: Cliente TCP (cabeceras y definición de constantes omitidas)

Asociar puerto a socket. En las líneas 14-17 se asocia el socket creado al puerto local que va a prestar el servicio. Para ello, primero se rellena la variable servaddr (estructura de direcciones IPv4, de tipo struct sockaddr_in). En concreto, se especifica familia Internet AF_INET, la dirección IP por la que se admitirán conexiones (INADDR_ANY, cualquiera de los interfaces de red) y el puerto local, pasado como parámetro desde la línea de comandos (argv[1]).

Convertir socket en socket a la escucha. En la línea 18 la función listen() convierte el socket en un socket a la escucha, es decir, ya preparado para aceptar conexiones entrantes. El segundo parámetro (LISTENQ) especifica el máximo número de conexiones pendientes que el kernel puede encolar.

Aceptar conexión del cliente. Normalmente, un proceso servidor pasa a la cola de bloqueados tras la llamada a la función accept() (línea 21), esperando a que llegue una petición de conexión de un cliente. Cuando se acepta dicha petición, accept() devuelve un nuevo descriptor de fichero que se usará para la comunicación con el cliente. Se crea un nuevo descriptor para cada cliente que se conecta al servidor.

Enviar mensaje al cliente. En la línea 24, se envía el mensaje al cliente mediante la función write().

```
1  int main(int argc, char **argv)
2  {
3      int listenfd, connfd;
4      struct sockaddr_in  servaddr = { 0 };
5      char buff[MAXLINE];
6      time_t ticks;
7
8      if (argc != 2)
9      {
10         printf("usage: %s <puerto>\n", argv[0]);
11         exit(0);
12     }
13     listenfd = socket(AF_INET, SOCK_STREAM, 0);
14     servaddr.sin_family      = AF_INET;
15     servaddr.sin_addr.s_addr = htonl(INADDR_ANY);
16     servaddr.sin_port        = htons(atoi(argv[1]));
17     bind(listenfd, (struct sockaddr *) &servaddr, sizeof(servaddr));
18     listen(listenfd, LISTENQ);
19     for ( ; ; )
20     {
21         connfd = accept(listenfd, (struct sockaddr *) NULL, NULL);
22         ticks = time(NULL);
23         snprintf(buff, sizeof(buff), "%.24s\n", ctime(&ticks));
24         write(connfd, buff, strlen(buff));
25         close(connfd);
26     }
27 }
```

Figura 2.4: Servidor TCP (cabeceras y definición de constantes omitidas)

Cerrar la conexión. La función close() en la línea 25 cierra la conexión con el cliente.

2.5.4. Ejecución de cliente y servidor TCP

Descarga los códigos, disponibles en *Moodle*. Se pueden descomprimir y desempaquetar con `tar xvzf RC_prac2.tar.gz` (eXtraer, Visualizar extracción, descomprimir Zip, Fichero RC_prac2.tar.gz). Compila cliente y servidor[2]:

```
$ gcc -o tcpcli tcpcli.c
$ gcc -o tcpsrv tcpsrv.c
```

A continuación, lanza el servidor en un puerto mayor que el 1024. Ten en cuenta que se quedará bloqueado esperando la petición de conexión de algún cliente:

```
$ ./tcpsrv 12345
```

Desde otro terminal, lanza el cliente indicando la dirección local y el puerto donde escucha el servidor:

```
$ ./tcpcli 127.0.0.1 12345
```

También puedes probar a conectarte a servidores de tus compañeros con el mismo comando, pero sustituyendo la dirección IP local (`127.0.0.1`) por la dirección IP de algún otro ordenador del laboratorio (`155.210.154.x`). Para que funcione, será necesario que en esa dirección haya un servidor escuchando.

La herramienta `strace` genera una traza de las llamadas al sistema realizadas por un proceso. Por ejemplo:

[2]El símbolo dólar representa la secuencia de caracteres que muestra el intérprete de comandos (*shell*) para indicar que está a la espera de órdenes.

```
struct addrinfo {
    int              ai_flags;       // AI_PASSIVE, AI_CANONNAME, etc.
    int              ai_family;      // AF_INET, AF_INET6, AF_UNSPEC
    int              ai_socktype;    // SOCK_STREAM, SOCK_DGRAM
    int              ai_protocol;    // use 0 for "any"
    size_t           ai_addrlen;     // size of ai_addr in bytes
    struct sockaddr  *ai_addr;       // struct sockaddr_in or _in6
    char             *ai_canonname;  // full canonical hostname
    struct addrinfo  *ai_next;       // linked list, next node
};
```

Figura 2.5: Estructura de (lista de) direcciones addrinfo

```
struct sockaddr_in {
    short int          sin_family;     // Address family, AF_INET
    unsigned short int sin_port;       // Port number
    struct in_addr     sin_addr;       // Internet address
    unsigned char      sin_zero[8];    // Same size as struct sockaddr
};
```

Figura 2.6: Estructura de dirección sockaddr_in (IPv4)

```
$ strace ./tcpsrv 12345
$ strace ./tcpcli 127.0.0.1 12345
```

Esta herramienta se puede utilizar para depurar código, ya que permite analizar los parámetros de las llamadas al sistema.

2.6. Estructuras de direcciones

La implementación de aplicaciones independientes del protocolo no es trivial, ya que cada protocolo tiene un formato de direcciones distinto. A continuación se describen las principales estructuras de datos del API de sockets. Tienes más detalles en el capítulo 3 de la *Guía de programación en red utilizando sockets*[3] disponible en Moodle.

La estructura addrinfo (figura 2.5) contiene una lista de direcciones. Cada una de estas direcciones, que puede corresponder a un protocolo distinto y por tanto tener detalles específicos, se almacena en el campo ai_addr. Es decir, el campo ai_addr, que formalmente es del tipo genérico struct sockaddr será en realidad de un tipo específico dependiendo de la familia de direcciones que corresponda. Las dos familias de direcciones que vamos a estudiar son las de Internet: struct sockaddr_in para IPv4 (figura 2.6, direcciones de 32 bits) y struct sockaddr_in6 para IPv6 (figura 2.7, direcciones de 128 bits).

Como en general no se conoce la familia de direcciones que usa el equipo remoto, todas las estructuras de direcciones usan los primeros 16 bits para indicar la familia a la que pertenecen. Así, se puede leer esa información independientemente del tipo de dirección y después interpretar el resto de la estructura de acuerdo a la familia de direcciones indicada. Como la estructura genérica struct sockaddr puede estar definida con un tamaño en el que no quepa una dirección IPv6, existe también la estructura genérica sockaddr_storage, que cumple la misma función pero con un tamaño mayor. En el caso de struct sockaddr_storage, los primeros 16 bits corresponden a un campo llamado ss_family. Así, se suele declarar una estructura sockaddr_storage y después usarla mediante interpretación explícita de tipos (*type casting*) como la estructura que interese.

[3] Beej's Guide to Network Programming Using Internet Sockets

```
1 struct sockaddr_in6 {
2     u_int16_t       sin6_family;    // address family, AF_INET6
3     u_int16_t       sin6_port;      // port number, Network Byte Order
4     u_int32_t       sin6_flowinfo;  // IPv6 flow information
5     struct in6_addr sin6_addr;      // IPv6 address
6     u_int32_t       sin6_scope_id;  // Scope ID
7 };
```

Figura 2.7: Estructura de dirección sockaddr_in6 (IPv6)

2.6.1. Implementación de *migetaddrinfo*

Para facilitar la tarea de construir la estructura de direcciones se usa la función getaddrinfo(). A esta función se le pasan como parámetros el nombre o dirección IP y el servicio o número de puerto de un equipo, y proporciona la estructura de direcciones con la información necesaria para crear un socket. Además, en la llamada se especifica con tanto detalle como se desee el tipo de dirección que se desee obtener.

En esta parte de la práctica hay que usar el programa del apartado 2.A (`migetaddrinfo.c`). Para poder compilar este código es necesario completar el código del apartado 2.B (`comun.c`). Este código realiza una llamada a getaddrinfo() e imprime la estructura de direcciones obtenida. Te será muy útil la información sobre la función getaddrinfo(), que puedes encontrar en el manual (`man getaddrinfo`) y en la sección 5.1 de la guía de programación en red utilizando sockets, disponible en Moodle.

Para completar el código, se recomienda seguir los siguientes pasos:

a) Completa los huecos numerados del código en el apartado 2.B

b) Realiza los cambios sobre el código que has descargado y descomprimido antes

c) Compila el código en un equipo del laboratorio y corrige posibles errores y avisos (*warnings*) de compilación

Puedes compilar el código con `gcc -Wall -Wextra -o migetaddrinfo migetaddrinfo.c comun.c`. La opciones de compilación `-Wall` y `-Wextra`, *warnings: all, extra*) son recomendables para que muestre todos los avisos, incluso los más triviales. Una vez el programa funcione correctamente, contesta a las siguientes preguntas:

4. ¿Qué *flag* hay que especificar en las pistas (hints) de la llamada a getaddrinfo() cuando vamos a solicitar una estructura de direcciones para lanzar un servidor?

5. Lanza `./migetaddrinfo www.unizar.es 80`. ¿Cuál es su dirección IP? Verifica que la salida del programa muestra el puerto 80 en formato local.

6. Desde tu equipo local, lanza `./migetaddrinfo moodle.unizar.es https`. ¿Cuál es su dirección IP? ¿Coincide el servicio https con el número de puerto que aparece en `/etc/services`?

7. ¿Qué ocurre al lanzar el programa especificando la dirección IP anterior y el número de puerto anterior?

8. Lanza `./migetaddrinfo www.v6.facebook.com http`. ¿En qué se diferencia la respuesta con respecto a los casos anteriores?

9. Ejecuta ahora `./migetaddrinfo hendrix-ssh.cps.unizar.es ssh`. ¿Qué puedes deducir?

10. Lanza `./migetaddrinfo www.google.com http`. ¿Qué puedes deducir?

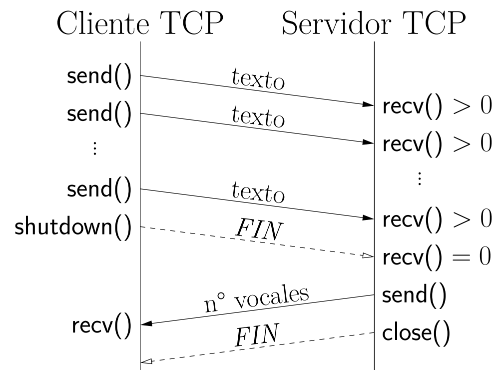


Figura 2.8: Interacción cliente-servidor en el protocolo cuenta-vocales

2.6.2. Preguntas de comprensión

Responde a las siguientes preguntas. No olvides que puedes consultar cualquier detalle de las llamadas en el manual (e.g. `man socket`).

11. Observa los parámetros que necesita la llamada socket() e indica a qué campos del struct addrinfo corresponden.

12. Observa los parámetros que necesita la llamada connect() e indica a qué campos del struct addrinfo corresponden.

13. ¿Es necesario que el servidor esté bloqueado esperando conexión de un cliente para poder crear el socket con la llamada socket() en el cliente? ¿Y para iniciar la conexión con la llamada connect()?

14. La llamada bind() asocia el socket con un puerto y es necesaria en el servidor. ¿Por qué?

15. En el servidor, la llamada accept() devuelve un descriptor de socket, con lo que en ese punto del programa disponemos de dos descriptores de socket: el devuelto por accept() y el devuelto por socket(). ¿Cuál de los dos utilizaremos en las llamadas send() y recv() del servidor?

16. Teniendo en cuenta los dos descriptores de socket anteriores, ¿cuál es la diferencia entre usar la función close() con cada uno de ellos?

2.7. Implementación de programas cliente/servidor

En las secciones 2.C y 2.D se presenta una pareja de programas cliente/servidor *incompletos*, también disponibles en *Moodle*. El cliente lee cadenas de caracteres de la entrada estándar y las envía al servidor. El servidor contabiliza el número de vocales que hay en los mensajes recibidos, y una vez terminados los envíos de cadenas, devuelve el número acumulado al cliente, que lo muestra por pantalla. Ten en cuenta que el cliente leerá hasta que se cierre su entrada estándar, lo cual está asociado a la combinación de teclas Ctrl + d . Es decir, cuando se pulsa Ctrl + d en el terminal donde se ejecuta el cliente, se cierra el fichero de entrada y finaliza el envío de datos al servidor. Como el cliente espera que el servidor envíe la cuenta de vocales, se usa la llamada al sistema shutdown(), que permite cerrar la conexión solamente en un sentido (en este caso de cliente a servidor), en lugar de close(), que cierra los dos sentidos de la conexión TCP.

La figura 2.8 muestra un esquema del protocolo.

En este apartado hay que completar los programas para que funcionen correctamente. Para ello, se propone seguir los siguientes pasos, empezando por el *cliente*:

a) Completa de forma esquemática los huecos numerados del código en la sección 2.C

b) Traslada el resultado anterior al código fuente

c) Compila el código y corrige los posibles errores *y avisos* de compilación

d) Para probar tu cliente TCP, tienes disponible un servidor TCP en lab000.cps.unizar.es, en el puerto 32000

A continuación, haz lo mismo para el *servidor*:

a) Completa de forma esquemática los huecos numerados del código en la sección 2.D

b) Traslada el resultado anterior al código fuente

c) Compila el código y corrige los posibles errores *y avisos* de compilación

d) Una vez que tu cliente TCP funcione correctamente, puedes utilizarlo para probar tu servidor

Cuando cliente y servidor funcionen correctamente, prueba lo siguiente.

5. Lanza el servidor en un puerto menor que 1024. ¿Qué error da?

6. En la misma máquina, lanza el servidor TCP en cierto puerto (mayor que 1024) y al mismo tiempo (desde otra ventana) lanza otro servidor TCP en el mismo puerto. ¿Es posible o da error?

2.8. Evaluación de la práctica

El trabajo realizado en esta práctica forma parte de la nota de prácticas de laboratorio de la asignatura. La entrega se realizará vía *Moodle* mediante un cuestionario donde se rellenarán los huecos de los códigos proporcionados. Es muy recomendable que antes de realizar el cuestionario te asegures de que tus códigos compilan correctamente (sin *warnings*) y funcionan como deberían. En la siguiente sesión de prácticas *necesitarás usar estos códigos*. ¡Ten en cuenta la *fecha límite* del cuestionario!

2.9. ¿Sabías que...?

- Además de en el fichero `/etc/services`, la correspondencia puertos-protocolos se puede consultar en:
  `http://en.wikipedia.org/wiki/List_of_TCP_and_UDP_port_numbers`.
- Un API (Application Program Interface) es el conjunto de funciones y procedimientos que ofrece cierta biblioteca para ser utilizado por otro software como una capa de abstracción. El interfaz de programación de aplicaciones de red original fue desarrollado para UNIX BSD (Universidad de Berkely). Para GNU/linux se denomina API de Sockets BSD o Sockets de Berkeley. Esta interfaz también se ha portado a Windows bajo el nombre Windows Sockets, abreviado como WinSock.
- TCP es uno de los protocolos fundamentales en Internet. Fue creado entre los años 1973 y 1974 por Vinton Cerf y Robert Kahn. Vinton Cerf fue investido Doctor Honoris Causa por la Universidad de Zaragoza en 2008 (ver detalles).

2.A. Código *migetaddrinfo* (`migetaddrinfo.c`)

```
1  /**
2   * @file migetaddrinfo.c
3   * Programa *migetaddrinfo* para construir estructuras de direcciones.
4   *
5   * Uso: migetaddrinfo [servidor] <puerto/servicio>
6   *
```

```
7    * Con dos argumentos, crea la estructura de direcciones correspondiente
8    * al servidor y servicio especificadosi y la imprime por pantalla.
9    * Con un argumento, crea la estructura de direcciones correspondiente
10   * al servicio especificado, en el equipo actual, y la imprime por pantalla.
11   */
12
13  // El preprocesador sustituye cada include por el contenido del fichero
14  // referenciado
15  // Bibliotecas estándar:
16  #include <stdio.h>   // printf()
17  #include <stdlib.h>  // exit()
18  #include <netdb.h>   // freeaddrinfo()
19
20  // Bibliotecas no estándar:
21  #include "comun.h"   // obtener_struct_direccion()
22
23  /**
24   * Programa principal migetaddrinfo.
25   *
26   * @param argc Número de argumentos usados en la línea de comandos.
27   * @param argv Vector de punteros a cadenas de caracteres. argv[0]
28   *             apunta al nombre del programa, argv[1] al primer
29   *             argumento y así sucesivamente.
30   * @return 0 si todo ha ido bien, distinto de 0 si hay error.
31   */
32  int main(int argc, char * argv[])
33  {
34      char f_verbose = 1;  // flag, 1: imprimir información por pantalla
35      struct addrinfo* direccion; // puntero (no inicializado!) a estructura de
            dirección
36
37      // verificación del número de parámetros:
38      if ((argc != 2) && (argc != 3))
39      {
40          printf("Número de parámetros incorrecto \n");
41          printf("Uso: %s [servidor] <puerto/servicio>\n", argv[0]);
42          exit(1); // finaliza el programa indicando salida incorrecta (1)
43      }
44
45      if (argc == 2)
46      {
47          // devuelve la estructura de dirección del servicio solicitado
48          // asumiendo que vamos a actuar como servidor
49          direccion = obtener_struct_direccion(NULL, argv[1], f_verbose);
50      }
51      else // if (argc == 3)
52      {
53          // devuelve la estructura de dirección al equipo y servicio solicitado
54          direccion = obtener_struct_direccion(argv[1], argv[2], f_verbose);
55      }
56
57      // cuando ya no se necesite, hay que liberar la memoria dinámica
58      // obtenida en getaddrinfo() mediante freeaddrinfo()
59      if (f_verbose)
60      {
61          printf("Devolviendo al sistema la memoria usada por servinfo (ya no se va a
                usar)... ");
62          fflush(stdout);
```

```
63      }
64      freeaddrinfo(direccion);
65      if (f_verbose) printf("hecho\n");
66      direccion = NULL;
67      // como ya se ha liberado ese bloque de memoria,
68      // dejamos de apuntarlo para evitar acceder a ella por error
69      // si referenciamos esta variable el programa abortará con
70      // ERROR: SEGMENTATION FAULT
71
72      // finaliza el programa indicando salida correcta (0)
73      exit(0);
74  }
```

2.B. Código común para gestionar direcciones (`comun.c`)

```
1   /**
2    * @file comun.c
3    *
4    * Este fichero contiene funciones comunes para trabajar con estructuras
5    * de direcciones.
6    */
7   #include <stdio.h>
8   #include <stdlib.h>
9   #include <string.h>
10  #include <sys/types.h>
11  #include <sys/socket.h>
12  #include <arpa/inet.h>
13  #include <netinet/in.h>
14  #include <netdb.h>
15
16  #include "comun.h"
17
18  /**
19   * Imprime una estructura *sockaddr_in* o *sockaddr_in6* almacenada en
20   * *sockaddr_storage*.
21   *
22   * @param saddr Estructura de dirección a imprimir.
23   */
24  void printsockaddr(struct sockaddr_storage * saddr)
25  {
26      struct sockaddr_in *saddr_ipv4; // puntero a estructura de dirección IPv4
27      // el compilador interpretará lo apuntado como estructura de dirección IPv4
28      struct sockaddr_in6 *saddr_ipv6; // puntero a estructura de dirección IPv6
29      // el compilador interpretará lo apuntado como estructura de dirección IPv6
30      void *addr; // puntero a dirección
31      // como puede ser tipo IPv4 o IPv6 no queremos que el compilador la
32      // interprete de alguna forma particular, por eso void
33      char ipstr[INET6_ADDRSTRLEN]; // string para la dirección en formato texto
34      int port; // almacena el número de puerto al analizar estructura devuelta
35
36      if (saddr == NULL)
37      {
38          printf("La dirección está vacía\n");
39      }
40      else
41      {
```

```
42          printf("\tFamilia de direcciones: ");
43          fflush(stdout);
44          if (saddr->ss_family == AF_INET6)
45          {   // IPv6
46              printf("IPv6\n");
47              // apuntamos a la estructura con saddr_ipv6 (cast evita warning),
48              // así podemos acceder al resto de campos a través de
49              // este puntero sin más casts
50              saddr_ipv6 = (struct sockaddr_in6 *)saddr;
51              // apuntamos al campo de la estructura que contiene la dirección
52              addr = &(saddr_ipv6->sin6_addr);
53              // obtenemos el puerto, pasando del formato de red al formato local
54              port = ntohs(saddr_ipv6->sin6_port);
55          }
56          else if (saddr->ss_family == AF_INET)
57          {   // IPv4
58              printf("IPv4\n");
59              saddr_ipv4 = [    1    ];
60              addr = [    2    ];
61              port = [    3    ];
62          }
63          else
64          {
65              fprintf(stderr, "familia desconocida\n");
66              exit(1);
67          }
68          // convierte la dirección ip a string
69          inet_ntop(saddr->ss_family, addr, ipstr, sizeof ipstr);
70          printf("\tDirección (interpretada según familia): %s\n", ipstr);
71          printf("\tPuerto (formato local): %d\n", port);
72      }
73  }
74
75
76  /**
77   * Función que devuelve un puntero a una estructura de direcciones rellenada
78   * con direcciones que cumplan los parámetros especificados.
79   *
80   * @param dir_servidor String con el nombre/dirección IP del equipo del
81   *                     que queremos obtener la dirección. Si está vacío
82   *                     (NULL) es que queremos la direccion del equipo local.
83   * @param servicio     String con el servicio/puerto a solicitar.
84   * @param f_verbose    Si es distinto de 0, se muestra información extra.
85   * @return Puntero a la estructura de direcciones creada en memoria dinámica.
86   */
87  struct addrinfo* obtener_struct_direccion(char *dir_servidor, char *servicio, char
        f_verbose)
88  {
89      struct addrinfo hints,     // Variable para especificar la solicitud
90                      *servinfo, // Puntero para respuesta de getaddrinfo()
91                      *direccion;// Puntero para recorrer la lista de
92                                 // direcciones de servinfo
93      int status;      // Finalización correcta o no de la llamada getaddrinfo()
94      int numdir = 1;  // Contador de estructuras de direcciones en la
95                       // lista de direcciones de servinfo
96
97      // sobreescribimos con ceros la estructura
98      // para borrar cualquier dato que pueda malinterpretarse
```

```
99      memset(&hints, 0, sizeof hints);
100
101     // genera una estructura de dirección con especificaciones de la solicitud
102     if (f_verbose)
103     {
104         printf("1 - Especificando detalles de la estructura de direcciones a
            solicitar... \n");
105         fflush(stdout);
106     }
107
108     // especificamos la familia de direcciones con la que queremos trabajar:
109     // AF_UNSPEC, AF_INET (IPv4), AF_INET6 (IPv6), etc.
110     hints.ai_family = [    4    ];
111
112     if (f_verbose)
113     {
114         printf("\tFamilia de direcciones/protocolos: ");
115         switch (hints.ai_family)
116         {
117             case AF_UNSPEC: printf("IPv4 e IPv6\n"); break;
118             case AF_INET:   printf("IPv4\n"); break;
119             case AF_INET6:  printf("IPv6\n"); break;
120             default:        printf("No IP (%d)\n", hints.ai_family); break;
121         }
122         fflush(stdout);
123     }
124
125     // especificamos el tipo de socket deseado:
126     // SOCK_STREAM (TCP), SOCK_DGRAM (UDP), etc.
127     hints.ai_socktype = [    5    ];
128
129     if (f_verbose)
130     {
131         printf("\tTipo de comunicación: ");
132         switch (hints.ai_socktype)
133         {
134             case SOCK_STREAM: printf("flujo (TCP)\n"); break;
135             case SOCK_DGRAM:  printf("datagrama (UDP)\n"); break;
136             default:          printf("no convencional (%d)\n", hints.ai_socktype);
                break;
137         }
138         fflush(stdout);
139     }
140
141     // flags específicos dependiendo de si queremos la dirección como cliente
142     // o como servidor
143     if (dir_servidor != NULL)
144     {
145         // si hemos especificado dir_servidor, es que somos el cliente
146         // y vamos a conectarnos con dir_servidor
147         if (f_verbose) printf("\tNombre/dirección del equipo: %s\n", dir_servidor);
148     }
149     else
150     {
151         // si no hemos especificado, es que vamos a ser el servidor
152         if (f_verbose) printf("\tNombre/dirección: equipo local\n");
153
```

```
154          // especificar flag para que la IP se rellene con lo necesario para hacer
                 bind
155          // consultar documentación con: 'man getaddrinfo')
156          hints.ai_flags = [     6     ];
157      }
158      if (f_verbose) printf("\tServicio/puerto: %s\n", servicio);
159
160      // llamada getaddrinfo() para obtener la estructura de direcciones solicitada
161      // getaddrinfo() pide memoria dinámica al SO,
162      // la rellena con la estructura de direcciones
163      // y escribe en servinfo la dirección donde se encuentra dicha estructura.
164      // La memoria dinámica reservada en una función NO se libera al salir de ella
165      // Para liberar esta memoria, usar freeaddrinfo()
166      if (f_verbose)
167      {
168          printf("2 - Solicitando la estructura de direcciones con getaddrinfo()... "
                 );
169          fflush(stdout);
170      }
171      status = getaddrinfo(dir_servidor, servicio, &hints, &servinfo);
172      if (status != 0)
173      {
174          fprintf(stderr,"Error en la llamada getaddrinfo(): %s\n", gai_strerror(
                 status));
175          exit(1);
176      }
177      if (f_verbose) printf("hecho\n");
178
179      // imprime la estructura de direcciones devuelta por getaddrinfo()
180      if (f_verbose)
181      {
182          printf("3 - Analizando estructura de direcciones devuelta... \n");
183          direccion = servinfo;
184          while (direccion != NULL)
185          {   // bucle que recorre la lista de direcciones
186              printf("    Dirección %d:\n", numdir);
187              printsockaddr((struct sockaddr_storage*) direccion->ai_addr);
188              // "avanzamos" a la siguiente estructura de direccion
189              direccion = direccion->ai_next;
190              numdir++;
191          }
192      }
193
194      // devuelve la estructura de direcciones devuelta por getaddrinfo()
195      return servinfo;
196  }
```

2.C. Código *cliente* de contar vocales (clientevocalesTCP.c)

```
1  /**
2   * @file clientevocalesTCP.c
3   *
4   * Programa *clientevocalesTCP* que envía cadenas de texto a un servidor.
5   *
6   * Uso: clientevocalesTCP servidor puerto
7   *
```

```
8   * El programa crea un socket TCP y lo conecta al servidor y puerto especificado.
9   * A través del socket envía cadenas de caracteres hasta llegar a fin de fichero
10  * (Control+d para provocarlo desde la entrada estándar).
11  * Finalmente, espera como respuesta el número total de vocales en las
12  * cadenas enviadas e imprime dicho valor por pantalla.
13  */
14
15 #include <stdio.h>
16 #include <unistd.h>
17 #include <stdlib.h>
18 #include <string.h>
19 #include <sys/types.h>
20 #include <sys/socket.h>
21 #include <arpa/inet.h>
22 #include <netinet/in.h>
23 #include <netdb.h>
24
25 #include "comun.h"
26
27 // definición de constantes
28 #define MAX_BUFF_SIZE 1000 ///< Tamaño del buffer para las cadenas de texto.
29
30 /**
31  * Función que crea la conexión y se conecta al servidor.
32  *
33  * @param servinfo Estructura de dirección a la que conectarse.
34  * @param f_verbose Flag.
35  * @return Descriptor de socket.
36  */
37 int initsocket(struct addrinfo *servinfo, char f_verbose)
38 {
39     int sock = -1;
40     int numdir = 1;
41
42     while (servinfo != NULL && sock < 0)
43     {   // bucle que recorre la lista de direcciones
44         printf("Intentando conexión con dirección %d:\n", numdir);
45
46         // crea un extremo de la comunicación y devuelve un descriptor
47         if (f_verbose)
48         {
49             printf("Creando el socket (socket)... ");
50             fflush(stdout);
51         }
52         sock = socket([ 7 ], [ 8 ], [ 9 ]);
53         if (sock < 0)
54         {
55             perror("Error en la llamada socket: No se pudo crear el socket");
56             // muestra por pantalla el valor de la cadena suministrada por el
57             // programador, dos puntos y un mensaje de error que detalla la
58             // causa del error cometido
59         }
60         else
61         {   // socket creado correctamente
62             if (f_verbose) printf("hecho\n");
63
64             // inicia una conexión en el socket:
65             if (f_verbose)
```

```
66              {
67                  printf("Estableciendo la comunicación a través del socket (connect)
                        ... ");
68                  fflush(stdout);
69              }
70              if (connect( 10 , 11 , 12 ) < 0)
71              {
72                  perror("Error en la llamada connect: No se pudo conectar con el
                        destino");
73                  close(sock);
74                  sock = -1;
75              }
76              else
77                  if (f_verbose) printf("hecho\n");
78          }
79
80          // "avanzamos" a la siguiente estructura de direccion
81          servinfo = servinfo->ai_next;
82          numdir++;
83      }
84
85      if (sock < 0)
86      {
87          perror("No se ha podido establecer la comunicación");
88          exit(1);
89      }
90
91      return sock;
92  }
93
94  /**
95   * Programa principal
96   *
97   * @param argc Número de argumentos usados en la línea de comandos.
98   * @param argv Vector de punteros a cadenas de caracteres. argv[0]
99   *             apunta al nombre del programa, argv[1] al primer
100  *             argumento y así sucesivamente.
101  * @return 0 si todo ha ido bien, distinto de 0 si hay error.
102  */
103 int main(int argc, char * argv[])
104 {
105     // declaración de variables propias del programa principal (locales a main)
106     char f_verbose = 1;         // flag, 1: imprimir información extra
107     struct addrinfo * servinfo; // puntero a estructura de dirección destino
108     int sock;                   // descriptor del socket
109     char msg[MAX_BUFF_SIZE];    // buffer donde almacenar datos para enviar
110     ssize_t len,        // número de bytes leídos por la entrada estándar
111             sentbytes, recvbytes;
112                         // número de bytes enviados/recibidos por el socket
113                         // (size_t con signo)
114     uint32_t num;       // variable donde anotar el número de vocales
115
116     // verificación del número de parámetros:
117     if (argc != 3)
118     {
119         printf("Número de parámetros incorrecto \n");
120         printf("Uso: %s servidor puerto/servicio\n", argv[0]);
121         exit(1); // finaliza el programa indicando salida incorrecta (1)
```

```
122     }
123 
124     // obtiene estructura de direccion
125     servinfo = obtener_struct_direccion(argv[1], argv[2], f_verbose);
126 
127     // crea un extremo de la comunicación con la primera de las
128     // direcciones de servinfo e inicia la conexión con el servidor.
129     // Devuelve el descriptor del socket
130     sock = initsocket(servinfo, f_verbose);
131 
132     // hay que liberar la memoria dinámica usada para la dirección
133     // cuando ya no se necesite
134     freeaddrinfo(servinfo);
135     servinfo = NULL;
136     // como ya se ha liberado ese bloque de memoria,
137     // dejamos de apuntarlo para evitar acceder a ella por error.
138     // Si referenciamos esta variable el programa abortará con
139     // ERROR: SEGMENTATION FAULT
140 
141     // bucle que lee texto del teclado y lo envía al servidor
142     printf("\nTeclea el texto a enviar y pulsa <Enter>, o termina con <Ctrl+d>\n");
143     while ((len = read(0, msg, MAX_BUFF_SIZE)) > 0)
144     {
145         // read lee del descriptor 0 (entrada estándar, por defecto el teclado)
146         // hasta que se pulsa INTRO,
147         // almacena en msg la cadena leída y
148         // devuelve la longitud de los datos leídos, en bytes
149         if (f_verbose) printf("  Leídos %zd bytes\n", len);
150 
151         // envía datos al socket
152         if ((sentbytes = send([   13   ], [   14   ], [   15   ], 0)) < 0)
153         {
154             perror("Error de escritura en el socket (send)");
155             exit(1);
156         }
157         else
158         {
159             if (f_verbose) printf("  Enviados %zd bytes al servidor\n",sentbytes);
160         }
161         // en caso de que el socket sea cerrado por el servidor,
162         // al llamar a send() se genera una señal SIGPIPE,
163         // que como en este código no se captura,
164         // hace que finalice el programa SIN mensaje de error
165         // Las señales se estudian en la asignatura Sistemas Operativos
166 
167         printf("Teclea el texto a enviar y pulsa <Enter>, o termina con <Ctrl+d>\n"
                 );
168     }
169 
170     // ya no queremos enviar más, cerramos para escritura
171     if (f_verbose)
172     {
173         printf("Cerrando el socket para escritura...");
174         fflush(stdout);
175     }
176     if (shutdown([   16   ], [   17   ]) < 0)
177     {
178         perror("Error al cerrar el socket para escritura (shutdown)");
```

```
179             exit(1);
180         }
181 
182         // el servidor verá la conexión cerrada y enviará el número de vocales
183         if (f_verbose)
184         {
185             printf("hecho\nEsperando respuesta del servidor...");
186             fflush(stdout);
187         }
188 
189         // recibe del servidor el número de vocales recibidas:
190         recvbytes = recv([   18   ], [   19   ], [   20   ], 0);
191         if (recvbytes != sizeof num)
192         {
193             printf("Recibidos %lu bytes en lugar de los %lu esperados", recvbytes,
                    sizeof num);
194             exit(1);
195         }
196         printf(" %ld bytes\n", sizeof num);
197         printf("Todo el texto enviado contenía en total %d vocales\n", [   21   ]);
198         // convierte el entero largo sin signo de formato de red a formato de host
199 
200         // cierra la conexión del socket
201         if (close([   22   ]) < 0)
202         {
203             perror("Error al cerrar el socket (close)");
204             exit(1);
205         }
206         else
207         {
208             if (f_verbose) printf("Socket cerrado\n");
209         }
210 
211         exit(0); // finaliza el programa indicando salida correcta (0)
212 }
```

2.D. Código *servidor* de contar vocales (servidorvocalesTCP.c)

```
 1 /**
 2  * @file servidorvocalesTCP.c
 3  *
 4  * Programa *servidorvocalesTCP* que cuenta vocales.
 5  *
 6  * Uso: servidorvocalesTCP puerto
 7  *
 8  * El programa crea un socket TCP en el puerto especificado, lo pone en modo
 9  * escucha y atiende consecutivamente a los clientes que se van conectando.
10  * Para cada cliente, recibe cadenas de texto, cuenta las vocales contenidas
11  * y al acabar todas las recepciones envía al cliente el número total de
12  * vocales contadas.
13  */
14 
15 #include <stdio.h>
16 #include <unistd.h>
17 #include <stdlib.h>
18 #include <string.h>
```

```
19 #include <sys/socket.h>
20 #include <arpa/inet.h>
21 #include <netdb.h>
22 #include <netinet/in.h>
23
24 #include "comun.h"
25
26 // definición de constantes
27 #define BUFF_SIZE 1000 ///< Tamaño del buffer para las cadenas de texto.
28
29
30 /**
31  * Función que crea la conexión y espera conexiones entrantes.
32  *
33  * @param servinfo Estructura de dirección local.
34  * @param f_verbose Flag.
35  * @return Descriptor de socket.
36  */
37 int establecer_servicio(struct addrinfo *servinfo, char f_verbose)
38 {
39     int sock = -1;
40
41     printf("\nSe usará ÚNICAMENTE la primera dirección de la estructura\n");
42
43     // crea un extremo de la comunicación y devuelve un descriptor
44     if (f_verbose)
45     {
46         printf("Creando el socket (socket)... ");
47         fflush(stdout);
48     }
49     sock = socket([    23    ], [    24    ], [    25    ]);
50     if (sock < 0)
51     {
52         perror("Error en la llamada socket: No se pudo crear el socket");
53         // muestra por pantalla el valor de la cadena suministrada por el
54         // programador, dos puntos y un mensaje de error que detalla la causa
55         // del error cometido
56         exit(1);
57     }
58     if (f_verbose) printf("hecho\n");
59
60     // asocia el socket con un puerto
61     if (f_verbose)
62     {
63         printf("Asociando socket a puerto (bind)... ");
64         fflush(stdout);
65     }
66     if (bind([    26    ], [    27    ], [    28    ]) < 0)
67     {
68         perror("Error asociando el socket");
69         exit(1);
70     }
71     if (f_verbose) printf("hecho\n");
72
73     // espera conexiones en un socket
74     if (f_verbose)
75     {
76         printf("Permitiendo conexiones entrantes (listen)... ");
```

```c
77          fflush(stdout);
78      }
79      listen([   29   ], 5);
80      // 5 es el número máximo de conexiones pendientes en algunos sistemas
81      if (f_verbose) printf("hecho\n");
82  
83      return sock;
84  }
85  
86  /**
87   * Función que cuenta las vocales en una cadena de texto
88   *
89   * @param msg Cadena con el mensaje al que contar vocales.
90   * @param s   Longitud del mensaje.
91   * @return Número de vocales en msg[0..s].
92   */
93  uint32_t countVowels(char msg[], size_t s)
94  {
95      uint32_t result = 0;
96      size_t i;
97      for (i = 0; i < s; i++)
98      {
99          if (msg[i] == 'a' || msg[i] == 'A' ||
100             msg[i] == 'e' || msg[i] == 'E' ||
101             msg[i] == 'i' || msg[i] == 'I' ||
102             msg[i] == 'o' || msg[i] == 'O' ||
103             msg[i] == 'u' || msg[i] == 'U') result++;
104     }
105     return result;
106 }
107 
108 /**
109  * Programa principal
110  *
111  * @param argc Número de argumentos usados en la línea de comandos.
112  * @param argv Vector de punteros a cadenas de caracteres. argv[0]
113  *             apunta al nombre del programa, argv[1] al primer
114  *             argumento y así sucesivamente.
115  * @return 0 si todo ha ido bien, distinto de 0 si hay error.
116  */
117 int main(int argc, char * argv[])
118 {
119     // declaración de variables propias del programa principal (locales a main)
120     char f_verbose = 1;          // flag, 1: imprimir información extra
121     struct addrinfo * servinfo; // dirección propia (servidor)
122     int sock, conn;              // descriptores de socket
123     char msg[BUFF_SIZE];         // espacio para almacenar los datos recibidos
124     ssize_t readbytes;           // numero de bytes recibidos
125     uint32_t num, netNum;        // contador de vocales en formato local y de red
126     struct sockaddr_storage caddr; // dirección del cliente
127     socklen_t clen = -1;         // longitud de la dirección
128 
129     // verificación del número de parámetros:
130     if (argc != 2)
131     {
132         printf("Número de parámetros incorrecto \n");
133         printf("Uso: %s puerto\n", argv[0]);
134         exit(1); // finaliza el programa indicando salida incorrecta (1)
```

```
135     }
136 
137     // obtiene estructura de direccion
138     servinfo = obtener_struct_direccion(NULL, argv[1], f_verbose);
139 
140     // crea un extremo de la comunicación. Devuelve el descriptor del socket
141     sock = establecer_servicio(servinfo, f_verbose);
142 
143     // hay que liberar la memoria dinámica usada para la dirección
144     // cuando ya no se necesite
145     freeaddrinfo(servinfo);
146     servinfo = NULL;
147     // como ya se ha liberado ese bloque de memoria,
148     // dejamos de apuntarlo para evitar acceder a ella por error.
149     // Si referenciamos esta variable el programa abortará con
150     // ERROR: SEGMENTATION FAULT
151 
152     // bucle infinito para atender conexiones una tras otra
153     while (1)
154     {
155         printf("\nEsperando conexión (pulsa <Ctrl+c> para finalizar la ejecución)
                ...\n");
156 
157         // acepta la conexión
158         clen = sizeof caddr;
159         if ((conn = accept([    30    ], (struct sockaddr *)&caddr, &clen)) < 0)
160         {
161             perror("Error al aceptar una nueva conexión (accept)");
162             exit(1);
163         }
164 
165         // imprime la dirección obtenida
166         printf("Aceptada conexión con cliente:\n");
167         printsockaddr(&caddr);
168 
169         // bucle de contar vocales
170         num = 0;
171         do {
172             if ((readbytes = recv([    31    ], [    32    ], BUFF_SIZE,0)) < 0)
173             {
174                 perror("Error de lectura en el socket (recv)");
175                 exit(1);
176             }
177             printf("Mensaje recibido del cliente: "); fflush(stdout);
178             write(1, msg, readbytes);
179             // muestra en pantalla (salida estándar 1) el mensaje recibido
180             // evitamos usar printf por si lo recibido no es texto acabado con \0
181             num += countVowels(msg, readbytes);
182             printf("Vocales contadas hasta el momento: %d\n",num);
183 
184         // condición de final: haber recibido 0 bytes (fin de fichero alcanzado)
185         } while (readbytes > 0);
186 
187         printf("\nSocket cerrado para lectura\n");
188         printf("Contadas %d vocales\n", num); // muestra las vocales recibidas
189         netNum = htonl(num); // convierte el entero largo sin signo hostlong
190         // desde el orden de bytes del host al de la red
191         // envia al cliente el número de vocales recibidas:
```

```
192          if (send([    33    ], &netNum, sizeof netNum,0) < 0)
193          {
194              perror("Error de escritura en el socket (send)");
195              exit(1);
196          }
197          if (f_verbose) printf("Enviado número de vocales contadas al cliente\n");
198
199          // cierra la conexión con el cliente
200          close([    34    ]);
201          if (f_verbose) printf("Cerrada la conexión con el cliente\n");
202      }
203
204      // código inalcanzable.
205      close(sock);
206      exit(0);
207  }
```

Práctica 3

Programación en red sobre UDP

3.1. Objetivos

Programación de una aplicación sencilla UDP, destacando las diferencias con respecto TCP.

3.2. Programación usando UDP

A partir de tu código TCP de la práctica anterior, realiza los cambios oportunos para que la comunicación entre cliente y servidor se haga usando UDP. Al igual que el servidor TCP, el servidor UDP debe ser capaz de servir a varios clientes, uno detrás de otro. Para ello sería necesario verificar si los mensajes recibidos son del mismo remitente, pero *no es necesario considerar el caso de varios remitentes simultáneos*. Aunque las llamadas al sistema a utilizar son ligeramente distintas (figura 3.1), los conceptos a aplicar son los mismos, con lo que te será útil toda la documentación de la práctica anterior. La figura 3.2 muestra un esquema del protocolo cuenta-vocales sobre UDP.

3.2.1. Detalles de implementación

Además de sustituir las llamadas al sistema, una transmisión mediante UDP es conceptualmente distinta a una mediante TCP. Esto implica que habrá que modificar otros aspectos del programa.

Uso de estructuras de direcciones

En el caso del cliente TCP, la estructura de direcciones solamente se usa al principio —`connect()`— y después se destruye —`freeaddrinfo()`—. Para UDP, el cliente necesitará la dirección del servidor en las llamadas `sendto()`.

En el caso del servidor TCP, la estructura de direcciones se obtiene al aceptar la conexión —`accept()`—, pero en UDP se obtendrá al recibir mensajes —`recvfrom()`—.

Imposibilidad de comprobar destino

Como en UDP no hay conexión, ya no se puede iterar la lista de direcciones hasta conseguir conectar. El cliente deberá elegir una dirección para el servidor y asumir que el servidor se encuentra en ella.

Marcado explícito de fin de envíos

En TCP se usa el propio estado de la conexión (abierta/cerrada en sentido cliente a servidor) para notificar al servidor que ya no se van a envíar más cadenas. Dado que en UDP no hay conexión, habrá que hacerlo mediante una notificación explícita de fin de transmisión. Es decir, al detectar «fin de fichero» el cliente deberá enviar una secuencia especial que no pueda confundirse con una cadena de texto. Como secuencia se usará un único byte con el número 4, que en ASCII simboliza el fin de transmisión. El servidor

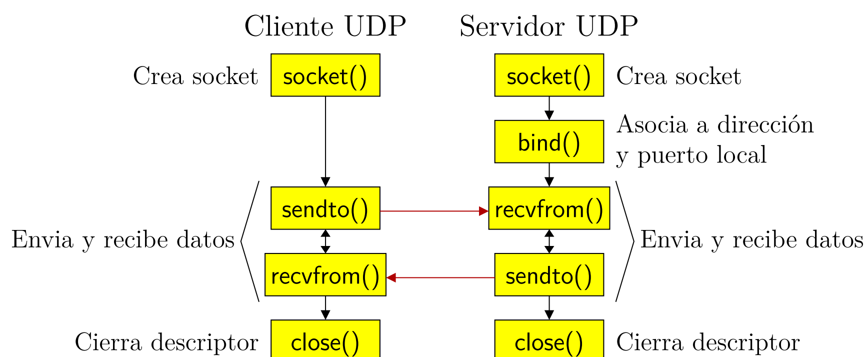


Figura 3.1: Llamadas al sistema para sockets con UDP

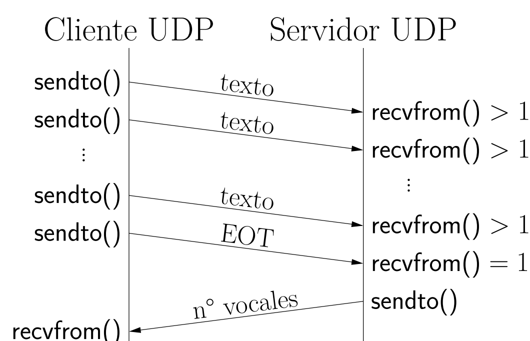


Figura 3.2: Interacción cliente-servidor en el protocolo cuenta-vocales sobre UDP

deberá verificar en cada recepción si ha recibido una cadena de texto o este byte con el número 4, y en ese caso asumir que la transmisión ha acabado y enviar el número de vocales. Tanto en el cliente como en el servidor puedes añadir la siguiente declaración para especificar el carácter de notificación de fin de transmisión.

```
1 const char fin = 4; // carácter ASCII end of transmission (EOT)
```

Limitación de tamaño de cada mensaje

Con UDP es importante tener en cuenta que cada envío genera un datagrama de tamaño limitado. Esto implica que hay que ser consciente del tamaño máximo de las tramas. En caso de existir algún problema relacionado con el tamaño del mensaje, el sistema lo notificaría con el error *EMSGSIZE* (`man sendto`). En el programa a realizar, dado que el tamaño máximo de las cadenas a enviar está limitado a 1 000 bytes, no debería haber ningún problema.

3.2.2. Servidor para pruebas

Tienes disponible un servidor UDP para hacer pruebas en lab000.cps.unizar.es, en el puerto 32000. Puedes utilizarlo para probar tu cliente UDP. Una vez que tu cliente UDP funcione correctamente, puedes utilizarlo para probar tu servidor.

3.2.3. Preguntas de comprensión

1. En la misma máquina, lanza el servidor UDP en cierto puerto (mayor que 1024) y al mismo tiempo (desde otra ventana) lanza otro servidor UDP en el mismo puerto. ¿Es posible o da error?

2. En la misma máquina, lanza el servidor TCP de la práctica anterior y al mismo tiempo (desde otra ventana) el servidor UDP de esta práctica. ¿Es posible o da error?

3. Teniendo en cuenta los resultados anteriores, ¿puede haber dos servidores usando el mismo protocolo de transporte (TCP o UDP) y el mismo puerto en la misma máquina? ¿Y si usan el mismo puerto pero uno usando TCP y otro usando UDP?

3.3. Automatización de la compilación con `make`

Cuando el número de ficheros fuente aumenta y/o contiene dependencias, es conveniente automatizar la compilación. Esta automatización compila los fuentes en el orden correcto. Además analiza las fechas de modificación de los ficheros y compila solo lo necesario. Una de las formas más habituales de automatizar la compilación es mediante el comando `make` y un fichero `Makefile`.

Un fichero `Makefile` básicamente contiene nombres de objetivos (usualmente el nombre de un fichero a generar), nombres de ficheros necesarios y comandos para alcanzar el objetivo. El objetivo y los ficheros necesarios deben estar en la misma línea, separados por «:». Los comandos van en las líneas siguientes, precedidos de un caracter tabulador. Para construir un objetivo hay que ejecutar `make objetivo`. Si se ejecuta `make` sin especificar un objetivo, se toma por defecto el objetivo *all*. Para los fuentes de las vocales UDP puedes utilizar el `Makefile` de la sección 3.A, disponible en Moodle.

4. ¿Qué hace el comando `make tarea`?

3.4. Capa de presentación

Una de las capas vistas en clase de teoría ha sido la *capa de presentación* de datos. Esta capa se encarga de homogeneizar los datos transmitidos entre distintos equipos. Como esta capa no existe en la arquitectura TCP/IP, este trabajo ha de hacerlo la aplicación. Ya hemos visto ejemplos de ello en las funciones tipo `ntohl()`, pero hay que verificar que cualquier dato sea interpretado correctamente. Por ejemplo, tanto en los equipos del laboratorio como en Hendrix, el texto se codifica mediante UTF-8.

5. Prueba a enviar desde el cliente al servidor de contar vocales los siguientes caracteres, cada uno en una línea distinta: línea sin ningún carácter, «a», «ñ», «€». ¿Cuántos bytes ocupa cada envío?

3.5. Evaluación de la práctica

El trabajo realizado en esta práctica forma parte de la nota de prácticas de laboratorio de la asignatura. Se puede trabajar en grupo, pero la práctica es *individual*: cada persona debe realizar y entregar su propio trabajo. Si el sistema de control antiplagio detecta un porcentaje significativo de similitud entre trabajos, puede conllevar la suspensión directa de las prácticas y del conjunto de la asignatura.

La entrega se realizará vía *Moodle* y consistirá en un único fichero `.tar.gz`. Este archivo debe incluir un fichero `Makefile` con las instrucciones para generar un cliente con nombre `clientevocalesUDP` y un servidor con nombre `servidorvocalesUDP` junto con el código fuente necesario para generar dichos ejecutables. Se permite organizar el código como se quiera, por ejemplo usar más/menos ficheros fuente, renombrarlos, modificar el `Makefile` proporcionado, etc. Tanto el cliente como el servidor deben compilar *sin avisos* (*warnings*) y deben funcionar correctamente en los equipos del laboratorio L1.02. La entrega debe realizarse antes de la fecha establecida en *Moodle*. *Entregas que no cumplan las especificaciones de la entrega, que no compilen, o que no funcionen obtendrán una nota de 0.*

3.6. ¿Sabías que...?

- El departamento de informática e ingeniería de sistemas dispone del equipo *lab000.cps.unizar.es*, equivalente a los del laboratorio, al que se puede acceder de forma remota. Puedes usar dicho equipo para probar o completar las tareas de prácticas desde fuera de la universidad.

3.A. `Makefile` para fuentes de contar vocales UDP

```
1  # DEFINICION DE VARIABLES Y CONSTANTES
2  CC=gcc
3  CFLAGS = -Wall -Wextra -Wshadow
4  UNAME := $(shell uname)
5  ifeq ($(UNAME), Linux) # equipos del laboratorio L1.02
6          LIBRARIES =
7  endif
8  ifeq ($(UNAME), SunOS) # Hendrix
9          LIBRARIES = -lsocket -lnsl
10 endif
11 
12 # OBJETIVOS
13 all: clientevocalesUDP servidorvocalesUDP
14 
15 clientevocalesUDP: clientevocalesUDP.o comun.o
16         $(CC) $(CFLAGS) -o clientevocalesUDP clientevocalesUDP.o comun.o $(
       LIBRARIES)
17 
18 servidorvocalesUDP: servidorvocalesUDP.o comun.o
19         $(CC) $(CFLAGS) -o servidorvocalesUDP servidorvocalesUDP.o comun.o $(
       LIBRARIES)
20 
21 clientevocalesUDP.o: clientevocalesUDP.c
22         $(CC) $(CFLAGS) -c clientevocalesUDP.c
23 
24 servidorvocalesUDP.o: servidorvocalesUDP.c
25         $(CC) $(CFLAGS) -c servidorvocalesUDP.c
26 
27 comun.o: comun.c
28         $(CC) $(CFLAGS) -c comun.c
29 
30 # objetivo para comprimir los fuentes tal y como se piden para la entrega
31 tarea:
32         tar cvzf "vocalesUDP_${LOGNAME}.tar.gz" *.c *.h ?akefile
33 
34 # objetivo para limpiar: borra todo lo generado
35 clean:
36         -rm -f clientevocalesUDP clientevocalesUDP.o servidorvocalesUDP
       servidorvocalesUDP.o comun.o "vocalesUDP_${LOGNAME}.tar.gz"
```

Práctica 4

Implementación de protocolos y algoritmos de secuenciación (trabajo práctico)

4.1. Objetivos

El objetivo de esta práctica es entender el trabajo práctico y empezar su implementación. El trabajo práctico consiste en implementar un protocolo de forma básica y extenderlo con los algoritmos de secuenciación de datos vistos en clase. Para ello hay que realizar un programa cliente que se comunique con el servidor proporcionado mediante el protocolo RCFTP.

4.2. Trabajo a realizar

Debes implementar *tres versiones* de un cliente que envíe su entrada estándar al servidor, que escribirá lo que reciba en el fichero `f_recibido`. El envío de datos debe cumplir las especificaciones del protocolo RCFTP, es decir, el cliente debe enviar los datos con el formato especificado en el protocolo y debe comprobar que el servidor confirma correctamente su recepción. El cliente debe estar preparado para trabajar tanto con IPv4 como con IPv6.

Invocado sin parámetros, el cliente muestra una ayuda con el formato que deben seguir sus parámetros:

```
Uso: ./rcftpclient [-v] -a[alg] [-t[Ttrans]] [-T[timeout]] [-w[tam]] -d<dirección>
-p<puerto>
  -v            Muestra detalles en salida estándar
  -a[alg]       Algoritmo de secuenciación a utilizar:
     1          Algoritmo básico
     2          Algoritmo Stop&Wait
     3          Algoritmo de ventana deslizante Go-Back-n
  -w[tam]       Tamaño (en bytes) de la ventana de emisión (sólo usado con -a3)
(por defecto: 2048)
  -t[Ttrans]    Tiempo de transmisión a simular, en microsegundos (por defecto: 200000)
  -T[timeout]   Tiempo de expiración a simular, en microsegundos (por defecto: 1000000)
  -d<dirección> Dirección del servidor
  -p<puerto>    Servicio o número de puerto del servidor
```

Las tres versiones del cliente son el algoritmo básico (sección 4.5), el algoritmo Stop&Wait (sección 4.6) y el algoritmo Go-back-n (sección 4.7). Para centrar la implementación en la programación del protocolo, puedes descargar de Moodle una versión incompleta del cliente a implementar. El cliente incompleto proporcionado contiene varias funciones ya implementadas (lectura de parámetros, lectura

de entrada estándar, visualización de dirección remota, visualización de estadísticas una vez finalizada la transmisión), así como el `Makefile` correspondiente para poder compilarlo. Además, probablemente puedas reutilizar parte del código que ya has realizado anteriormente en el cliente-servidor UDP. Concretamente, en el fichero comprimido del cliente incompleto encontrarás los siguientes ficheros:

- rcftp.h/.c: Cabeceras y funciones del protocolo RCFTP
- multialarm.h/.c: Cabeceras y funciones de gestión de temporizadores
- rcftpclient.h/.c: Cabeceras, programa principal y varias funciones del cliente RCFTP
- Makefile: Dependencias y comandos para compilar mediante `make` (o `gmake`)
- misfunciones.h: Definiciones y cabeceras de varias de las funciones a implementar
- misfunciones.c: Espacio para implementar las funciones que necesites

4.3. Protocolo RCFTP

El programa de contar vocales implementado anteriormente realiza un intercambio de datos muy simple, donde los datos no necesitan ninguna organización particular. En aplicaciones más complejas los datos transmitidos necesitan una organización más estructurada, especificada por un protocolo. Para implementar el programa cliente de esta práctica es imprescindible conocer el protocolo de comunicaciones que va a utilizar: RCFTP. Dicho protocolo ha de ser usado tanto por el cliente como por el servidor.

RCFTP es un protocolo que, usado sobre UDP, añade fiabilidad y secuenciación de datos. En nuestro caso, el cliente enviará mensajes de acuerdo a las especificaciones correspondientes y el servidor le irá confirmando la recepción correcta de los mensajes.

4.3.1. Formato del mensaje

El protocolo RCFTP especifica el formato de las peticiones del cliente y respuestas del servidor (ambas de tamaño fijo) como se detalla a continuación:

1 B	1 B	2 B	4 B	4 B	2 B	512 B
Versión	Flags	Sum	NúmSeq	Next	Len	Buffer

```
1 struct rcftp_msg {
2     uint8_t  version; /**< Versión RCFTP_VERSION_1; otro valor es inválido */
3     uint8_t  flags;   /**< Flags. Máscara de bits de los defines F_X */
4     uint16_t sum;     /**< Checksum en network format calculado con xsum */
5     uint32_t numseq;  /**< Número de secuencia, medido en bytes */
6     uint32_t next;    /**< Siguiente numseq esperado, medido en bytes */
7     uint16_t len;     /**< Longitud de datos válidos en el campo buffer */
8     uint8_t  buffer[RCFTP_BUFLEN]; /**< Datos de tamaño fijo 512 B */
9 };
```

Cada campo almacena la siguiente información:

- Versión (1 byte): contiene el número 1, que identifica la versión del protocolo. Nos permitirá cambiar nuestro protocolo y seguir identificando paquetes de implementaciones más viejas.
- Flags (1 byte): contiene la máscara de bits de los *defines* `F_X`. Ver la sección 4.3.2
- Checksum (2 bytes): contiene la negación (inversión de bits) de la suma *complemento a uno* de la petición/respuesta. Esta operación ya está implementada en la función `xsum()` que tienes en el código proporcionado. Como el checksum siempre se calcula/verifica con respecto al mensaje en formato de red, no hay que aplicar ninguna función de reordenación de bytes sobre él.

- Número de secuencia (4 bytes en la ordenación de bytes de la red): contiene el número de secuencia del primer byte de datos que contiene el mensaje. El primero de los mensajes debe tener 0 como número de secuencia.
- Next/ACK o siguiente número de secuencia esperado (4 bytes en la ordenación de bytes de la red): contiene el número de secuencia del siguiente byte de datos que esperamos recibir. Ten en cuenta que el cliente no espera recibir datos.
- Longitud (2 bytes en la ordenación de bytes de la red): contiene el número de bytes de datos válidos que hay en el campo *buffer*.
- Datos o buffer: contiene los datos y es siempre de longitud RCFTP_BUFLEN.

4.3.2. Flags

Los *flags* del protocolo RCFTP especifican el tipo de petición del cliente o estado del servidor como se detalla a continuación:

- `F_NOFLAGS`: Flag por defecto.
- `F_BUSY`: Flag de servidor ocupado atendiendo a otro cliente.
- `F_FIN`: Flag de intención/confirmación de finalizar transmisión.
- `F_ABORT`: Flag de aviso de finalización forzosa.

Los mensajes generados por el cliente siempre deben tener el valor `F_NOFLAGS`, excepto cuando se transmita el último mensaje de datos, en cuyo caso se usará el valor `F_FIN` en los flags. El cliente deberá terminar cuando reciba la confirmación de ese último mensaje (también marcado con el flag `F_FIN`) por parte del servidor.

4.4. Servidor RCFTPd

El código del servidor *rcftpd* (RCFTP dæmon) está disponible en Moodle en un fichero comprimido, con los ficheros fuente estructurados igual que en el cliente. Se usa de la forma siguiente:

```
Uso: ./rcftpd -p<puerto> [-v] [-a[alg]] [-e[frec]] [-t[Ttrans]] [-r[Tprop]]
  -p<puerto>    Especifica el servicio o número de puerto
  -v            Muestra detalles en salida estándar
  -a[alg]       Ajusta el comportamiento al algoritmo del cliente (por defecto: 0):
      0:        Sin mensajes incorrectos
      1:        Fuerza mensajes incorrectos hasta su corrección
      2:        Fuerza mensajes incorrectos/pérdidas/duplicados hasta su corrección
      3:        Fuerza mensajes incorrectos/pérdidas/duplicados
  -e[frec]      Fuerza en media un mensaje incorrecto de cada [frec] (por defecto: 5)
  -t[Ttrans]    Tiempo de transmisión a simular, en microsegundos (por defecto: 200000)
  -r[Tprop]     Tiempo de propagación a simular, en microsegundos (por defecto: 250000)
Nota: Los algoritmos 1-3 generan errores aleatoriamente. Además, los algoritmos 1 y 2
mantienen el error generado hasta que el cliente responda correctamente.
```

Ten en cuenta que gran parte de la implementación del cliente ya está hecha en el servidor, con lo que es muy conveniente estudiar el servidor antes de programar el cliente.

Abre el fichero `rcftpd.h` y observa que ahí figuran todas las funciones, con una breve descripción de su funcionamiento. Si lo consideras oportuno, puedes copiarlas en tu cliente y adaptarlas como necesites.

Abre el fichero `rcftpd.c` y localiza el código de la función «process_requests». Dicha función implementa el funcionamiento general del servidor, es decir, recibe mensajes, los procesa, construye mensajes de respuesta, planifica su envío y finalmente los envía. Hay varios puntos especialmente interesantes en el código:

- Localiza el comentario «construir el mensaje válido» y observa que un mensaje se construye simplemente rellenando la estructura rcftp_msg, que se enviará posteriormente.
- Observa que el parámetro de entrada en la función que calcula checksum (xsum()) es la propia estructura rcftp_msg, es decir, el checksum se calcula sobre el contenido de esa estructura. Eso implica que la estructura debe rellenarse *antes* de calcular su checksum.
- Ten en cuenta también que el campo sum forma parte de la propia estructura, y que la función xsum() tendrá en cuenta su valor. Observa el código para ver cómo se consigue que el contenido de ese campo no afecte al resultado.

Como habrás observado, las funciones asociadas al protocolo RCFTP han sido separadas explícitamente de las del servidor RCFTPd para poder usarlas directamente desde el cliente. Abre el fichero `rcftp.h`. Observa que la declaración del struct rcftp_msg es ligeramente distinta de la del enunciado de la práctica. A través de una directiva de preprocesador, el código determina si está siendo compilado por GCC o no, y dependiendo de esa condición se decide qué línea de código hay que usar. En la línea de código específica para GCC, el código tiene todos los campos del struct con atributo «packed» para especificar al compilador GCC que los ponga todos contiguos en memoria. Dicho atributo no pertenece al lenguaje C, sino al «dialecto de C» que entiende GCC, con lo que el resto de los compiladores generarían un error en ese punto. Aún sin poner ese atributo, el struct está organizado de forma que un compilador no se vea tentado a poner huecos entre campos. Aún así, podría darse el caso de que, si el compilador intentara optimizar mucho, pusiera cada campo en una dirección de memoria múltiplo del tamaño en bytes de sus registros (por ejemplo alineando todo a 64 bits). Eso podría ser más rápido en ejecución, porque el procesador podría cargar/guardar cada campo en memoria sin tener que alinear, pero sería desastroso para el programa, puesto que estaríamos enviando estructuras que no coincidirían con las especificadas en el protocolo. Es importante saber que los compiladores toman muchas decisiones que van más allá del lenguaje de programación. Al no estar presentes en el lenguaje, si un programador quiere especificar cómo tomar alguna de estas decisiones, debe recurrir a atributos, opciones de compilación, o directivas específicas del compilador de que se trate.

Observa las cabeceras y comentarios del fichero `rcftp.h`.

1. ¿Qué funciones tienes ya implementadas en el fichero `rcftp.c`?
2. ¿Hay que aplicar htons() al usar xsum()?

Tanto el código del cliente como el del servidor incluyen un fichero `Makefile`, de forma que para compilar basta con invocar el comando `make` (`gmake` en *Hendrix*). Este comando contiene las instrucciones para compilar y enlazar cada uno de los ficheros fuente con las bibliotecas necesarias, tanto en GNU/Linux-x86_64 (L1.02) como en SunOS-sparc (Hendrix). Visualiza el fichero `Makefile` del cliente para ver su funcionamiento.

3. ¿Qué hace el objetivo *tarea* en el cliente?

4.4.1. Traza del servidor

A continuación puedes ver un ejemplo de lo que muestra por pantalla el servidor, con la opción `-v`, al comunicarse con un cliente que funciona. Tras implementar tu cliente, el servidor debería mostrar una salida similar.

```
Fichero "f_recibido" abierto para escritura
rcftpd $Revision$
Servidor RCFTP en puerto 23456
Comunicación con el puerto 55496 del equipo 127.0.0.1 usando IPv4

Mensaje RCFTP recibido:
     Versión: 1
```

```
      Flags: 0
      Núm. secuencia: 0
      Next: 0
      Len: 512
      Checksum (net): 0x8a43 (ok)
Planificando envío 0 (todo correcto)

Realizando envio 0 (todo correcto)
Mensaje RCFTP enviado:
      Versión: 1
      Flags: 0
      Núm. secuencia: 0
      Next: 512
      Len: 0
      Checksum (net): 0xc1cd (ok)

Mensaje RCFTP recibido:
      Versión: 1
      Flags: 0
      Núm. secuencia: 512
      Next: 0
      Len: 512
      Checksum (net): 0x5589 (ok)
Planificando envío 1 (todo correcto)

Realizando envio 1 (todo correcto)
Mensaje RCFTP enviado:
      Versión: 1
      Flags: 0
      Núm. secuencia: 0
      Next: 1024
      Len: 0
      Checksum (net): 0xbfcd (ok)

Mensaje RCFTP recibido:
      Versión: 1
      Flags: 0
      Núm. secuencia: 1024
      Next: 0
      Len: 512
      Checksum (net): 0x214 (ok)
Planificando envío 2 (todo correcto)

Realizando envio 2 (todo correcto)
Mensaje RCFTP enviado:
      Versión: 1
      Flags: 0
      Núm. secuencia: 0
      Next: 1536
      Len: 0
      Checksum (net): 0xbdcd (ok)

Mensaje RCFTP recibido:
      Versión: 1
      Flags: 0
      Núm. secuencia: 1536
      Next: 0
      Len: 512
      Checksum (net): 0x3745 (ok)
Planificando envío 3 (checksum con bytes desordenados y recepción incorrecta)

Realizando envio 3 (checksum con bytes desordenados y recepción incorrecta)
Mensaje RCFTP enviado:
      Versión: 1
      Flags: 0
```

```
      Núm. secuencia: 0
      Next: 1536
      Len: 0
      Checksum (net): 0xcdbb (error)

Mensaje RCFTP recibido:
      Versión: 1
      Flags: 0
      Núm. secuencia: 1536
      Next: 0
      Len: 512
      Checksum (net): 0x3745 (ok)
Planificando envío 4 (todo correcto)

Realizando envio 4 (todo correcto)
Mensaje RCFTP enviado:
      Versión: 1
      Flags: 0
      Núm. secuencia: 0
      Next: 2048
      Len: 0
      Checksum (net): 0xbbcd (ok)

Mensaje RCFTP recibido:
      Versión: 1
      Flags: 0
      Núm. secuencia: 2048
      Next: 0
      Len: 512
      Checksum (net): 0xa39c (ok)
Planificando envío 5 (checksum incorrecto y recepción incorrecta)

Realizando envio 5 (checksum incorrecto y recepción incorrecta)
Mensaje RCFTP enviado:
      Versión: 1
      Flags: 0
      Núm. secuencia: 0
      Next: 2048
      Len: 0
      Checksum (net): 0xbacd (error)

Mensaje RCFTP recibido:
      Versión: 1
      Flags: 0
      Núm. secuencia: 2048
      Next: 0
      Len: 512
      Checksum (net): 0xa39c (ok)
Planificando envío 6 (todo correcto)

Realizando envio 6 (todo correcto)
Mensaje RCFTP enviado:
      Versión: 1
      Flags: 0
      Núm. secuencia: 0
      Next: 2560
      Len: 0
      Checksum (net): 0xb9cd (ok)

Mensaje RCFTP recibido:
      Versión: 1
      Flags: 0
      Núm. secuencia: 2560
      Next: 0
      Len: 512
```

```
      Checksum (net): 0x5aa3 (ok)
Planificando envío 7 (todo correcto)

Realizando envio 7 (todo correcto)
Mensaje RCFTP enviado:
      Versión: 1
      Flags: 0
      Núm. secuencia: 0
      Next: 3072
      Len: 0
      Checksum (net): 0xb7cd (ok)

Mensaje RCFTP recibido:
      Versión: 1
      Flags: 2
      Núm. secuencia: 3072
      Next: 0
      Len: 306
      Checksum (net): 0xe60b (ok)
Planificando envío 8 (todo correcto)

Realizando envio 8 (todo correcto)
Mensaje RCFTP enviado:
      Versión: 1
      Flags: 2
      Núm. secuencia: 0
      Next: 3378
      Len: 0
      Checksum (net): 0xb699 (ok)
Flag F_FIN recibido y confirmado
Recepción finalizada. Compara los ficheros para verificar los datos recibidos.
```

Como puedes observar, el servidor muestra todos los mensajes recibidos y enviados, indicando además si son correctos o no.

Con el objetivo de simular el retardo de red, el servidor no realiza los envíos inmediatamente, sino que espera cierto tiempo antes de enviar (dependiendo de los tiempos de transmisión y propagación introducidos como parámetros). Es decir, para cada mensaje recibido, el servidor *planifica* un envío, *identificado con un número*. Después de los retardos correspondientes, ese envío planificado será realmente enviado (y recibido por el cliente inmediatamente por estar muy cercanos cliente y servidor).

Como ayuda para la depuración, la salida está coloreada utilizando códigos ANSI. En caso de que vuelques la salida en un fichero para analizarlo posteriormente, para que los códigos de color se interpreten correctamente puedes volcar ese fichero por pantalla usando `cat` o puedes usar el comando `less` con la opción `-R`.

4.5. Algoritmo básico

Se utilizará el parámetro `-a1` tanto en el cliente como en el servidor. El programa cliente debe asumir que todos los mensajes enviados tendrán respuesta. El cliente deberá verificar que la respuesta sea válida y sea la esperada (la confirmación de recepción correcta de lo enviado por el cliente). Si es así, el cliente deberá enviar un nuevo mensaje con el siguiente bloque de datos. En caso contrario, el cliente deberá reenviar el mensaje anterior.

Puedes ver el pseudocódigo de la descripción anterior en el algoritmo 1. ¡Es esencial que entiendas el pseudocódigo antes de empezar a programarlo!

4.5.1. Ayuda a la implementación del algoritmo básico

Como ayuda a la implementación, ten en cuenta lo siguiente:

Algorithm 1 Algoritmo básico RCFTPClient

```
ultimoMensaje ← false
ultimoMensajeConfirmado ← false
datos ← leerDeEntradaEstandar(RCFTP_BUFLEN)
if finDeFicheroAlcanzado then
   ultimoMensaje ← true
end if
mensaje ← construirMensajeRCFTP(datos)
while ultimoMensajeConfirmado = false do
   enviar(mensaje)
   recibir(respuesta)
   if esMensajeValido(respuesta) and esLaRespuestaEsperada(respuesta) then
      if ultimoMensaje then
         ultimoMensajeConfirmado ← true
      else
         datos ← leerDeEntradaEstandar(RCFTP_BUFLEN)
         if finDeFicheroAlcanzado then
            ultimoMensaje ← true
         end if
         mensaje ← construirMensajeRCFTP(datos)
      end if
   end if
end while
/*** La función construirMensajeRCFTP() debe rellenar cada uno de los campos del mensaje (struct rcftp_msg) a enviar ***/
/*** La función esMensajeValido() debe comprobar la versión y el checksum del mensaje ***/
/*** La función esLaRespuestaEsperada() debe comprobar que respuesta.next sea mensaje.numseq+mensaje.len, que no haya flags de «ocupado/abortar» en respuesta, y que si mensaje.flags tiene marcado «fin» tambin lo tenga respuesta.flags ***/
```

- El código proporcionado como plantilla para el cliente está organizado en archivos de forma similar al servidor. En principio sólo necesitas editar los archivos `misfunciones.c` y `misfunciones.h`, aunque puedes editar también el resto del código.

- Ya tienes implementada la lectura de parámetros de la línea de comandos, la lectura de datos desde la entrada estándar y una función que muestra información sobre la comunicación tras finalizarla correctamente.

- El establecimiento del socket UDP (initsocket()) y las funciones de manejo de estructuras de direcciones (obtener_struct_direccion(), printsockaddr()) son las que ya has implementado en prácticas anteriores; copia/pega/ajusta lo necesario.

- El código proporcionado compila sin errores (aunque sí con *warnings*, por estar incompleto). Una forma de trabajar es compilar después de cada cambio introducido y verificar que no se ha introducido nada que genere errores de compilación.

- No uses funciones de *strings* para procesar los datos a transmitir, porque esos datos no tienen porqué ser cadenas de caracteres.

- En la compilación verás que se compilan también los fichero `multialarm.c` y `vemision.c`. De momento puedes ignorarlo, ya que en el algoritmo básico no se utilizan.

- Empieza la implementación del cliente asumiendo que no habrá errores en la comunicación (puedes lanzar el servidor con `-a0` para ello), centrándote así en aspectos de la comunicación.

- Cuando el cliente funcione correctamente asumiendo una comunicación sin errores, pasa a probarlo con errores (`-a1` en el servidor) y verifica caso por caso todos los errores que puedan presentarse. ¡No asumas que los campos del mensaje recibido son correctos! ¡Comprueba que contienen lo esperado!

4.5.2. Ayuda para probar el algoritmo básico

Para probar el funcionamiento del algoritmo básico, ten en cuenta lo siguiente:

- Para enviar un fichero, utiliza una tubería en la entrada estándar: `cat fichero | ./rcftpclient -v -a1 -d<servidor>-p<puerto>`

- Se recomienda transmitir un fichero con datos aleatorios. Puedes generar un fichero de 20 kB con contenido aleatorio mediante el comando:
`dd if=/dev/urandom of=mificheroaleatorio bs=1000 count=20`

- Puedes comprobar que el fichero enviado coincide con el fichero `f_recibido` generado por el servidor usando el comando `cmp mificheroaleatorio f_recibido`

- Al principio se recomienda lanzar el servidor en el equipo del laboratorio (que es donde se evaluará la entrega), en el puerto 32002 (otros puertos pueden estar filtrados por cortafuegos), y el cliente en el mismo equipo. Cuando funcione se recomienda probar a lanzar el cliente en Hendrix. Recuerda que los equipos del laboratorio tienen procesadores muy distintos a los de Hendrix, con lo que no sirven los mismos ejecutables ni los mismos ficheros objeto generados durante la compilación. Haz `make clean` (o `gmake clean`) antes de compilar en otro equipo

- Puedes utilizar *Wireshark* para monitorizar los paquetes que viajan por la red

4.5.3. Preguntas sobre el funcionamiento

4. Dado que el servidor nunca envía datos, ¿qué habrá en el campo `next` del mensaje enviado por el cliente?

5. Dados los tiempos de transmisión y propagación que simula el servidor, asumiendo que no hay errores, ¿cuánto debería costar la transferencia de un fichero de 20 kB con el algoritmo básico?

6. Lanza el servidor sin errores (`-a0`) y anota cuánto tiempo tarda tu cliente en transferir un fichero de 20 kB. ¿Se parece al tiempo anterior?

7. Lanza ahora el servidor con errores (`-a1`) y anota cuánto tiempo tarda tu cliente (`-a1`) en transferir el mismo fichero. Ten en cuenta que, si no has especificado lo contrario, el servidor genera un mensaje incorrecto de cada cinco. ¿Cuánto ha tardado la transferencia en este caso? ¿qué relación tiene con respecto al tiempo sin errores?

4.6. Algoritmo *Stop&Wait*

Esta versión se lanzará mediante el parámetro `-a2`, tanto en el cliente como en el servidor RCFTP. Para esta versión hay que añadir el tratamiento de pérdidas de mensajes al algoritmo básico.

Para implementar este algoritmo es necesario utilizar tiempos de expiración o *timeouts*. Una opción es utilizar la señal de alarma, tal y como se ha visto en la asignatura *Sistemas Operativos*. No obstante, la opción recomendada es utilizar el código *multialarm* proporcionado. La siguiente sección explica cómo usarlo.

4.6.1. Gestión de timeouts mediante *multialarm*

El código *multialarm* proporciona una interfaz sencilla para el manejo de múltiples alarmas simultáneamente, aunque para el algoritmo *Stop&Wait* sólo se necesita una. Además, al añadir un timeout bloquea momentáneamente al proceso, simulando el *tiempo de transmisión* que requeriría el envío asociado al timeout.

- Antes de programar timeouts hay que especificar (una sola vez) su duración y la duración del tiempo de transmisión, en microsegundos, mediante la función settimeoutduration(). Esto ya está hecho en la plantilla proporcionada a partir de los parámetros introducidos por línea de comandos.
- Antes de usar los temporizadores de *multialarm* hay que asociar la rutina de servicio de la alarma a la función *int handle_sigalrm(int signal)* mediante:

```
signal(SIGALRM, handle_sigalrm);
```

- La función void addtimeout() programa un timeout que vencerá después de los microsegundos indicados en la función anterior.
- La función void canceltimeout() cancela el próximo timeout a vencer.
- La función int getnumtimeouts() devuelve el número de timeouts pendientes de vencer[1].
- La variable *volatile int timeouts_vencidos* almacena en todo momento el número de timeouts vencidos. Ten en cuenta que esta variable *volatile* se modifica desde la rutina de servicio, con lo que para evitar posibles inconsistencias *no hay que modificarla* desde el programa. La forma más sencilla para actuar ante el vencimiento de los timeouts es comparar esta variable con otra (e.g. *timeouts_procesados*) inicializada a 0 y, si son distintas, tratar el timeout (salir del bucle de espera) e incrementar en uno la otra variable (ver algoritmo 2).

4.6.2. Interrupciones de llamadas al sistema

Ten en cuenta que un timeout (señal de alarma) puede vencer mientras el programa está bloqueado en una llamada al sistema (e.g. read(), recvfrom()). Si esto sucede en *Hendrix*, la llamada acabará con resultado -1 y error EINTR. GNU/Linux no presenta este comportamiento, y la llamada al sistema *no es interrumpida*. Para conseguir el mismo comportamiento en ambos sistemas, la opción recomendada es configurar el socket como «no bloqueante». Con esta configuración, si recvfrom() (o read()) no tiene datos que recibir, devolverá -1 y error EAGAIN en lugar de bloquearse esperando datos. Para poner el socket en modo «no bloqueante» se puede usar el siguiente código:

```
int sockflags;
sockflags = fcntl(socket,F_GETFL,0);     //obtiene el valor de los flags
fcntl(socket, F_SETFL, sockflags | O_NONBLOCK); //modifica el flag de bloqueo
```

4.6.3. Ayuda a la implementación del algoritmo *Stop&Wait*

Como ayuda a la implementación, ten en cuenta lo siguiente:

- El código cliente proporcionado ya incluye todo lo necesario para usar *multialarm*.
- Si vence un timeout, seguramente no habrás recibido el mensaje esperado. ¡No verifiques la corrección de mensajes que no se han recibido!
- Funcionalmente, lo único que cambia respecto al algoritmo básico *al enviar* es que además de enviar hay que añadir un timeout.
- Funcionalmente, lo único que cambia respecto al algoritmo básico *al recibir* es que hay que esperar una respuesta *o el vencimiento de un timeout*. El algoritmo 2 detalla este comportamiento.

[1] En Stop&Wait nunca va a haber más de un timeout pendiente de vencer.

Algorithm 2 Pseudocódigo que sustituye a «recibir(respuesta)» en el algoritmo básico

```
addtimeout()
esperar ← true
while esperar do
    numDatosRecibidos ← recibir(respuesta) /*** No bloqueante: devuelve -1 si no hay datos ***/
    if numDatosRecibidos> 0 then
        canceltimeout() /*** Se puede realizar después de comprobar que es el mensaje esperado ***/
        esperar ← false
    end if
    if timeouts_procesados ≠ timeouts_vencidos then
        esperar ← false
        timeouts_procesados ← timeouts_procesados+1
    end if
end while
/*** comprobar respuesta SOLO si realmente se ha recibido; reenviar mensaje en caso contrario ***/
```

4.6.4. Preguntas sobre el funcionamiento

De forma similar a como habías hecho con el algoritmo básico, lanza el servidor con errores (`-a2`) y anota cuánto tiempo tarda tu cliente en transferir un fichero del mismo tamaño usando *Stop&Wait* (`-a2`).

8. Compáralo con los tiempos del algoritmo básico (`-a1` en cliente y servidor). ¿Con *Stop&Wait* y pérdidas tarda más o menos? ¿Por qué?

9. Lanza ahora tu cliente *Stop&Wait* (`-a2`) con errores básicos en el servidor (`-a1`) y compáralo con lo que tarda el algoritmo básico de tu cliente (`-a1`). ¿Qué puedes deducir?

10. Finalmente, compara cuánto se tarda en transferir el mismo fichero con tus dos versiones del cliente (`-a1`, `-a2`) con el servidor sin errores (`-a0`). ¿Tiene sentido?

4.7. Algoritmo *Go-Back-n*

Para esta versión, el programa cliente deberá usar el algoritmo de ventana deslizante *Go-Back-n*. Se utilizará el parámetro `-a3` en el cliente, y debe funcionar tanto con `-a2` como con `-a3` en el servidor RCFTP. El parámetro `-w[bytes]` indicará el tamaño de la ventana de emisión a utilizar, en bytes. Es importante destacar que ciertos detalles de *Go-Back-n* pueden implementarse de formas distintas, es decir, no hay una única implementación correcta. En la realidad, cada sistema operativo implementa su versión particular de TCP, que incorpora una versión adaptada de ventana deslizante. Por ejemplo, en tu implementación deberás tomar varias decisiones:

- Cuando el cliente reciba una respuesta, ¿debe comprobar si es un duplicado de la anteriormente recibida y en ese caso ignorar esa respuesta, o debe tratar cada mensaje de forma independiente?
- Cuando el cliente reciba una respuesta incorrecta, ¿debe asumir que el mensaje enviado se ha perdido y reenviarlo, o debe simplemente ignorar la respuesta incorrecta y no reenviar hasta que venza el timeout?
- Cuando el cliente reciba una respuesta incorrecta, ¿debe reenviar obligatoriamente los mensajes posteriores que haya en su ventana, o debe dar cierto margen por si son confirmados?
- etc.

Dependiendo de las decisiones que tomes, la transmisión será más o menos eficiente, más o menos rápida, le afectarán más o menos ciertos errores, etc. Independientemente de los detalles anteriores, la implementación debería tener un esquema general como el mostrado en el algoritmo 3.

Algorithm 3 Esquema general del cliente con ventana deslizante

```
while ultimoMensajeConfirmado = false do
    /*** BLOQUE DE ENVIO: Enviar datos si hay espacio en ventana ***/
    if espacioLibreEnVentanaEmision and finDeFicheroNoAlcanzado then
        datos ← leerDeEntradaEstandar()
        mensaje ← construirMensajeRCFTP(datos)
        enviar(mensaje)
        addtimeout()
        añadirDatosAVentanaEmision(datos)
    end if
    /*** BLOQUE DE RECEPCION: Recibir respuesta y procesarla (si existe) ***/
    numDatosRecibidos ← recibir(respuesta) /*** No bloqueante: devuelve -1 si no hay datos ***/
    if numDatosRecibidos > 0 then
        if esMensajeValido(respuesta) and esRespuestaEsperadaGBN(respuesta) then
            canceltimeout()
            liberarVentanaEmisionHasta(respuesta.next)
            if esConfirmacionDeUltimosDatos(respuesta) then
                ultimoMensajeConfirmado ← true
            end if
        end if
    end if
    /*** BLOQUE DE PROCESADO DE TIMEOUT ***/
    if timeouts_procesados ≠ timeouts_vencidos then
        mensaje ← construirMensajeMasViejoDeVentanaEmision()
        enviar(mensaje)
        addtimeout()
        timeouts_procesados ← timeouts_procesados+1
    end if
end while
/*** La función esRespuestaEsperadaGBN() debe comprobar que respuesta.next−1 esté dentro de la
ventana de emisión y que no haya flags de «ocupado/abortar» en respuesta ***/
```

4.7.1. Gestión de timeouts e interrupciones

El algoritmo *Go-Back-n* necesita poder manejar varios timeouts simultáneamente. Para ello, puedes utilizar el código *multialarm* (sección 4.6.1). Además, este algoritmo necesita que el cliente no se bloquee al realizar `recvfrom()`s (o `read()`s) cuando no haya datos que recibir. Este comportamiento se puede conseguir de varias formas (hilos de ejecución, funciones `select()`, `poll()`, etc.). La recomendación para esta práctica es configurar el socket como «no bloqueante» (sección 4.6.2), ya que te permitirá reutilizar en gran medida el código de los algoritmos anteriores.

4.7.2. Ayuda a la implementación del algoritmo *Go-Back-n*

- Se recomienda haber completado el algoritmo *Stop&Wait* antes de pasar al *Go-Back-n*.
- Es importante verificar que el cliente sigue funcionando si se especifica `-a1` y `-a2` en el servidor. ¡Si el cliente *Go-Back-n* funciona con `-a3` en el servidor, pero no con `-a2` en el servidor, es que el cliente no funciona correctamente!
- Ten en cuenta que el cliente enviará varios mensajes sin esperar confirmaciones, con lo que una confirmación no estará asociada necesariamente al último mensaje enviado. Eso implica que habrá que crear una nueva función `esRespuestaEsperadaGBN()` para trabajar con ventana.
- Puedes probar el funcionamiento de la ventana de emisión lanzando el cliente antes que el servidor,

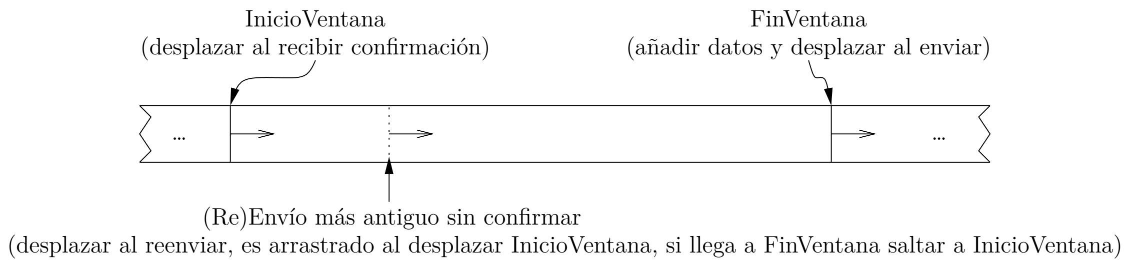


Figura 4.1: Esquema de funcionamiento de la ventana de emisión

o parando el servidor durante una transmisión. En ambos casos el cliente debería reenviar secuencialmente el contenido de su ventana una y otra vez, y el total de datos enviados debería ser igual al tamaño de ventana especificado como parámetro.

- Puedes usar el código proporcionado en `vemision.c/.h`, que implementa una cola circular estática en un vector (tal y como has visto en las asignaturas *Programación II* y *Estructuras de datos y algoritmos*) con un índice adicional para reenvíos (ver figura 4.1).

4.7.3. Preguntas sobre el funcionamiento

11. Lanza el servidor con errores (`-a3`) y anota cuánto tiempo tarda tu cliente en transferir un fichero del mismo tamaño que en las pruebas anteriores usando *Stop&Wait* (`-a2`). Repite la operación utilizando el cliente con *Go-Back-n* (`-a3`) y compara el tiempo necesario para transferir el fichero. ¿Qué puedes deducir?

12. Lanza otra vez ambos con `-a3`, pero ahora especificando en el cliente un timeout de 0,5 s. ¿Qué sucede en la comunicación?

13. Lanza otra vez ambos con `-a3`, pero ahora especificando en el cliente un timeout de 5 s. ¿Qué sucede en la comunicación?

14. Compara ahora la comunicación (ambos con `-a3`) con un servidor con $T_t = 50$ ms (pon también este valor en el cliente) y $T_p = 300$ ms, frente a otra comunicación con el servidor con $T_t = 300$ ms (pon también este valor en el cliente) y $T_p = 50$ ms. ¿Cuál funciona mejor? ¿Por qué?

15. Calcula el valor óptimo de la ventana para los tiempos por defecto ($T_t = 200$ ms, $T_p = 250$ ms) y pásalo a unidades de byte según el tamaño del mensaje RCFTP. Lanza una comunicación con un valor de ventana (`-w`) la mitad del valor anterior (aproximadamente), y otra comunicación con un valor de ventana el doble del anterior (aproximadamente). ¿Qué puedes deducir?

4.8. Evaluación del trabajo

El trabajo deberá entregarse antes de la fecha *límite* especificada a través de Moodle. Antes de la fecha límite se pueden realizar entregas *recomendadas* del trabajo a través de Moodle (¡revisa las fechas concretas!). Posteriormente a cada fecha de entrega recomendada, se evaluarán los trabajos entregados y se informará a los autores de los resultados. En caso de que se encuentren fallos, puedes aprovechar para corregirlos y entregar una nueva versión del trabajo antes de la fecha límite.

Recuerda que puedes acudir a tu profesor para resolver dudas del trabajo. No obstante, ten en cuenta que es un trabajo de redes de computadores y no de programación. Los profesores de la asignatura no vamos a evaluar aspectos de programación *ni a resolver dudas sobre programación*. Sólo se atenderán consultas relativas a la comprensión del trabajo a realizar y los algoritmos para realizarlo, es decir, *sin revisar el código*.

Al realizar la entrega habrá que tener en cuenta los siguientes puntos:

1. Hay que comprimir todos los fuentes necesarios para compilar el *cliente* (¡no el servidor!) en un único fichero. Los ficheros a comprimir deberán estar en un fichero `tar` comprimido mediante `gzip` (con extensión resultante `.tar.gz`). Recuerda que el fichero `Makefile` incluye un objetivo para generar este fichero.
2. Entre los ficheros comprimidos debe encontrarse un `Makefile`, a través del cual el programa debe *compilar sin errores* en el sistema GNU/Linux de los equipos del laboratorio y debe funcionar tal y como se indica en el enunciado.
3. El ejecutable debe imprimir por pantalla el nombre del autor o autores (máximo 2).
4. No es imprescindible que las entregas tengan implementados todos los algoritmos (básico, *Stop&Wait*, *Go-Back-n*).
5. Para cada fecha de entrega, se permiten *múltiples reenvíos* de la práctica, es decir, si después de enviarla se detecta algún error, se puede volver a enviar. Se evaluará únicamente la última versión enviada.
6. En caso de no realizar la entrega límite, se obtendrá la nota de la última entrega recomendada, si existe.
7. La evaluación del trabajo incluye un *control anticopia* que compara cada trabajo con otros trabajos entregados (incluyendo los de convocatorias y cursos anteriores). Si el porcentaje de similitud de dos trabajos supera cierto umbral se calificarán con 0.

Si no se cumple algún punto, se podrá reducir la nota obtenida.

4.9. ¿Sabías que...?

- Existen herramientas para generar automáticamente la documentación de código. Para ello, los comentarios introducidos en el código deben seguir cierto formato. En el código de la práctica, el código se ha comentado de forma que la herramienta `doxygen` pueda generar su documentación. Esta herramienta se puede usar sobre una gran variedad de lenguajes de programación (C, C++, Java, C#, PHP, Fortran, VHDL, etc.) y permite generar documentación en múltiples formatos, tales como HTML, XML, RTF, páginas de `man`, PostScript y PDF (vía LaTeX).
- Este trabajo es una versión *extremadamente simplificada* de uno de los trabajos que se pedían en la asignatura «Laboratorio de Computadores», en el antiguo plan de Ingeniería Informática. Para esa asignatura no se proporcionaba nada del código del cliente (ahora dispones del `Makefile`, del main(), de las funciones para leer parámetros y datos a transmitir, de un módulo para gestionar múltiples alarmas, y ya has implementado las funciones initsocket(), obtener_struct_direccion() y printsockaddr() en prácticas anteriores. Además, en el trabajo original se pedía específicamente tratar mensajes duplicados, no habían visto los algoritmos de ventana deslizante en clases de teoría y no se les proporcionaban los pseudocódigos a implementar. Finalmente, además del código del cliente, se les pedía una memoria con la documentación de todo el proceso de diseño del cliente, problemas encontrados, justificación de soluciones adoptadas, pruebas de rendimiento, etc.

Práctica 5

Topología y tráfico en Internet

5.1. Objetivos

Exploración de la topología y la organización del tráfico en Internet. Uso de herramientas comunes de exploración de redes.

5.2. Historia

ARPANET (*Advance Research Projects Agency NETwork*) fue la precursora de Internet. En una primera fase, en 1969, conectó cuatro nodos ubicados en los campus de Los Ángeles (UCLA) y Santa Bárbara (UCSB) de la Universidad de California, la Universidad de Utah y el *Augmentation Research Center* en el *Stanford Research Institute*. Sirvió para validar el modelo de conmutación de paquetes y que las citadas instituciones compartieran computadores muy costosos en aquella época. De ahí que una de sus primeras aplicaciones fuera `telnet`, un comando para conectarse a equipos remotos como el actual `ssh` pero sin cifrado.

Al año siguiente ARPANET se extendió a la costa este de EE.UU. y un poco más tarde, en 1973, llegó a Europa gracias a un enlace satelital que la conectó con Noruega y el Reino Unido. En el año 1980 apareció *Usenet*, una red temática de intercambio de opiniones, bromas o trucos de programación que es considerada como la primera red social.

En 1984 ARPANET se convirtió en Internet. Los operadores se dieron cuenta de que una red de tales dimensiones era complicada de administrar. La red debía ser organizada de manera descentralizada «como una red de redes». De esta manera, cada red sería gestionada por organizaciones diferentes pero asegurando su conectividad con el resto de redes mediante el cumplimiento de estándares abiertos. Una de las primeras redes en conectarse fue la *Computer Science NETwork* (CSNET) fundada por la *National Science Foundation* (NSF) que unía los departamentos de informática de los campus de todo EEUU. La NSF creó numerosos centros de computación en todo el país. Para conectarlos creó la primera red troncal, la NSFNET, una infraestructura de alta velocidad que permitía conectar todos sus centros. En aquellos lugares donde no tenían acceso a NSFNET, se organizaban regionalmente para conectarse entre ellas y al punto más próximo de la NSFNET. Su uso se extendió a otros centros de investigación y posteriormente a organizaciones comerciales.

5.3. Topología y tráfico en Internet

La estructura de Internet en cuanto a topología parece un caos de redes y enlaces que crecen de forma continua en todas partes y sin ningún control. Sin embargo, a pesar del desorden aparente, se puede encontrar una cierta estructura jerárquica. Los proveedores de servicios de Internet (ISP, *Internet Service Providers*) son empresas que proporcionan servicios para acceder, usar o participar en la red Internet. Estas empresas pueden clasificarse de acuerdo a los servicios que ofrecen a sus clientes. Hay

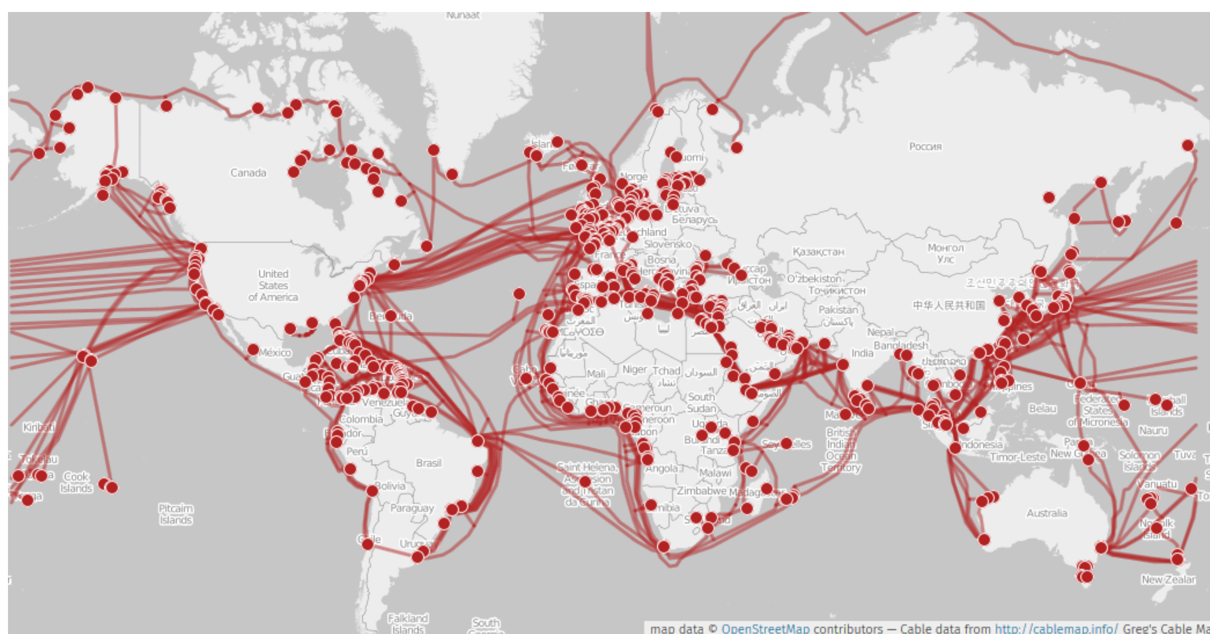


Figura 5.1: Mapa mundial de cables submarinos de comunicaciones en 2015 🅭🅯🄎 2.0. Datos de cables por Greg Mahlknecht (GPLv3, 2015-07-21). Mapa mundial por contribuyentes Openstreetmap.

ISPs pequeños que tienen en propiedad una red de alcance local o regional, y contratan líneas a otras empresas para poder ofrecer el servicio de acceso a Internet. También pueden encontrarse los ISP de nivel nacional, que cuentan con una infraestructura de red a nivel del país (o grupo de países, como España y Portugal) en que desarrollan su actividad. Estos ISPs de nivel nacional tendrán que contratar líneas o servicios a terceras empresas cuando quieran ofrecer una «salida» internacional a sus clientes, mientras que el tráfico nacional puede ir por su red. Finalmente existen grandes compañías de telecomunicaciones que cuentan con líneas propias para conectar sus infraestructuras en diversos países. En este caso, es usual que dichas compañías se presenten como distintos ISPs (distintas marcas o filiales). Entre ellas están las empresas que cuentan con las líneas continentales e intercontinentales.

1. Visita `https://www.submarinecablemap.com` para visualizar una versión interactiva actualizada del mapa de la figura 5.1.

Un *sistema autónomo* (SA/AS) es un grupo de redes IP que tienen una política de encaminamiento común. Un SA es *multihomed* si está conectado a varios SAs, y es de *tránsito* si permite el tráfico entre los SAs a los que está conectado. Un SA *stub* está conectado solamente a un SA de tránsito.

En general, una empresa con grandes infraestructuras propias, sea ISP o no, tiene registrados uno o más SAs, cada uno con su *Autonomous System Number* (ASN). La asignación de ASNs es realizada por los *Registros Regionales de Internet* (RIR). Por ejemplo, en *RIPE NCC* puede encontrarse toda la información sobre los SAs que gestiona. Como dicha información es enorme, es más cómodo filtrarla para las necesidades específicas que se requieran. En este caso nos interesa el buscador RIPEstat (`https://stat.ripe.net/ui2013/`).

2. Escribe «rediris» en el campo de búsqueda y espera que muestre elementos relacionados (*no hagas clic en «Go»*). La red académica y de investigación española RedIRIS es un SA. ¿Cuál es su ASN? ¿Coincide con el que figura en las transparencias de teoría?

3. ¿Cuál es el ASN del SA GEANT? ¿Coincide con el que figura en las transparencias de teoría?

4. ¿Cuántos SAs contienen «Vodafone» en su nombre? Al menos uno de dichos SAs debería ser de *tránsito*.

5. Grandes compañías que no están dedicadas a tránsito suelen tener SAs *multihomed*. ¿Cuántos SAs tiene *Nestle*?

6. Empresas o instituciones no tan grandes también pueden llegar a tener un SA de tipo *stub*, es decir, conectado únicamente a SAs de tránsito. ¿Tiene SA el Boletín Oficial del Estado (BOE)?

Vuelve a introducir «rediris», y haz clic en su ASN (no en «Go») para obtener información detallada de dicho sistema autónomo. Por ejemplo, en el recuadro «Routing Status» se indica el valor de sus *Observed BGP neighbours*. Puedes hacer clic en ese valor para ver sus vecinos.

7. ¿Cuántos vecinos BGP tiene RedIRIS?

8. ¿Cuántos vecinos BGP tiene el BOE? ¿Cuáles son?

5.3.1. Tráfico entre SAs

La red Internet se basa en el modelo de conmutación de paquetes, por lo que la información fluye en forma de datagramas. Dado que existen bastantes SAs interconectados, cada uno de estos datagramas podría seguir un camino totalmente diferente al seguido por los demás. Sin embargo, el funcionamiento de Internet se rige en gran medida por patrones económicos: la mayoría de redes (SAs) pertenecen a empresas, que negocian con otras empresas las rutas a utilizar. Es decir, para establecer rutas, una empresa paga por usar las redes de otras empresas, y a su vez cobra por el uso de su propia red por parte de otras empresas.

Para que una empresa pueda usar las infraestructuras de otras se debe establecer una relación previa, de forma que la empresa propietaria de las líneas reciba una compensación por el tráfico que cursa. A esta relación se le denomina «acuerdo de tránsito». Estos acuerdos se reflejan en la configuración BGP de cada SA, tal y como has comprobado en preguntas anteriores. Existe otro tipo de relación entre empresas que se produce cuando las dos empresas son más o menos de igual tamaño y el tráfico que intercambian es similar. En estos casos, se crean acuerdos de intercambio de datos sin que exista contraprestación económica. A estos acuerdos se los denomina «acuerdos de *peering*», ya que se establecen entre empresas que se reconocen como iguales o *peers*. Debido a estos acuerdos, el camino seguido por un paquete desde su origen hasta su destino es más predecible de lo que se pudiese pensar en un principio y no cambia prácticamente de un paquete a otro a no ser que algún encaminador esté muy congestionado o fuera de servicio.

Esta relación jerárquica entre ISPs puede dar lugar a problemas de retardo cuando dos usuarios ubicados en zonas cercanas pero con ISPs diferentes desean intercambiar datos. En este caso, los paquetes de datos podrían circular por enlaces internacionales en su viaje, a pesar de la cercanía geográfica entre origen y destino. Por ejemplo, los paquetes cursados en la conexión entre dos usuarios leoneses A y B que usan los ISPs locales *Nora* y *Lesein* podrían seguir el siguiente itinerario:

> Usuario A → Nora Internet (León, España) → Retevisión (España) → BT Ignite (España) → BT Ignite (Reino Unido) → BT Ignite (EE.UU.) → Cable & Wireless (EE.UU.) → Cable & Wireless (Amsterdam, Holanda) → Cable & Wireless (España) → Telefónica Data (España) → Lesein (León, España) → Usuario B

Como puede observarse, los datos se intercambian en Estados Unidos, donde existe un acuerdo de tránsito (o de *peering*) entre *BT Ignite* y *Cable & Wireless*, por lo que, para conectar a dos usuarios locales, los datos viajan hasta Estados Unidos a través de las redes de seis ISPs diferentes. Esta situación, además de causar un mayor retardo en las comunicaciones, saturaba los enlaces transoceánicos (que son menos numerosos que los continentales), por lo que los ISPs buscaron una forma de evitar el problema. La solución fue crear los llamados *puntos neutros* de intercambio, cuya función es conectar a los proveedores de una zona geográfica para evitar que el tráfico con origen y destino en dicha zona salga al exterior de la misma.

En España, el punto neutro a nivel nacional se denomina *Espanix* y está en Madrid. El punto neutro es un lugar físico con equipos de comunicaciones que permiten intercambiar los datos entre los ISPs que así lo desean. Para poder conectarse a Espanix, un ISP debe tener medios suficientes para dirigir el

tráfico internacional por sí mismo, sin que intervenga ninguna de las redes de los otros ISPs conectados. Una vez conectados los ISPs, el intercambio de datos efectivo se debe consensuar entre cada par de ISPs, mediante acuerdos de *peering*.

9. Consulta en la web de Espanix (`http://www.espanix.net`) cuáles son sus socios.

10. ¿Se encuentra RedIRIS entre los socios de Espanix?

11. ¿Coincide el ASN de RedIRIS con el hallado anteriormente?

En España existen otros puntos neutros de ámbito geográfico más restringido, como por ejemplo *EuskoNIX* y *CATNIX*, que se encargan de mantener el tráfico local dentro del País Vasco y Cataluña, respectivamente. También hay puntos neutros en Europa. La *European Internet Exchange Association* (`http://www.euro-ix.net`) es una asociación de puntos neutros europeos.

12. Observa la lista de ASNs conectados a los puntos neutros europeos (IXPDB → ASN Directory → # of IXP connection). ¿Qué empresas están presentes en más puntos neutros?

13. Visita la web de Hurricane Electric, `http://www.he.net`, y visualiza el mapa 3D de su red (HE 3D Network Map).

14. En cuanto a latencia, ¿qué implicaciones tiene para un proveedor estar en muchos puntos neutros?

5.3.2. Consultas mediante WHOIS

Además de los datos proporcionados por los distintos RIRs y puntos neutros, se puede consultar información adicional mediante el protocolo WHOIS. Puedes realizar consultas con este protocolo mediante el comando `whois`. Para especificar la base de datos a consultar, usa la opción `-h`.

Usa el comando `whois -h whois.ripe.net` con la dirección IPv4 de tu equipo del laboratorio (155.210.154.x). Prueba a no indicar ninguna base de datos o a indicar una base de datos distinta como `whois -h whois.afrinic.net` y observa las diferencias.

15. ¿Cuál es el nombre de la red a la que pertenece (*netname*)?

16. ¿A qué SA pertenece dicha red (*mnt-by*)?

17. ¿Cuál es su ASN (*origin*)?

18. ¿Qué responde `whois -h whois.arin.net` en *Comment* al usar la dirección IP 127.0.0.1? ¿Coincide con lo visto en clase de teoría?

19. ¿Y al preguntar por las direcciones IP 10.0.0.1 y 192.168.1.2? Puedes consultar RFC1918 para más información.

20. Averigua mediante `whois` a qué SA pertenece la dirección IP pública que te proporciona el ISP en tu casa.

21. Pregunta por el ASN de RedIRIS (`whois as<núm>`) y observa que se muestran sus acuerdos de tránsito. Comprueba si EASYNET se encuentra entre los SAs desde los que acepta tráfico (*import/from/accept*). ¿Cuál es su ASN? Comprueba si este SA está en Espanix.

22. Entre los SAs hacia los que anuncia sus propias rutas (*export/to/announce*), ¿se encuentra AS15169? ¿A qué empresa pertenece ese SA?

Habrás notado que las respuestas pueden incluir direcciones postales y personas responsables de las distintas redes.

23. Usa `whois` para preguntar por la persona con identificador `MJG8-RIPE`. ¿De qué red crees que es responsable?

El comando `whois` instalado en los equipos no permite consultas IPv6, aunque dichas consultas son válidas. Para probarlo, accede al servidor de búsquedas de ARIN (`https://search.arin.net/`).

24. Observa que en la parte superior de la página muestra un enlace con la dirección IPv6 de tu equipo y haz clic sobre ella para buscar esa dirección mediante WHOIS en ARIN. ¿A qué compañía corresponde?

5.4. Herramienta `ping`

La herramienta `ping` es quizá la más básica de todas las herramientas de red. Tiene como finalidad el comprobar si determinado equipo es alcanzable y es capaz de responder a nuestros mensajes. Ejemplo:

```
lab000:~$ ping www.google.com
PING www.l.google.com (74.125.39.106) 56(84) bytes of data.
64 bytes from fx-in-f106.1e100.net (74.125.39.106): icmp_seq=1 ttl=51 time=37.1 ms
64 bytes from fx-in-f106.1e100.net (74.125.39.106): icmp_seq=2 ttl=51 time=40.2 ms
64 bytes from fx-in-f106.1e100.net (74.125.39.106): icmp_seq=3 ttl=51 time=37.1 ms
64 bytes from fx-in-f106.1e100.net (74.125.39.106): icmp_seq=4 ttl=51 time=37.0 ms
64 bytes from fx-in-f106.1e100.net (74.125.39.106): icmp_seq=5 ttl=51 time=40.9 ms
^C
--- www.l.google.com ping statistics ---
5 packets transmitted, 5 received, 0% packet loss, time 4533ms
rtt min/avg/max/mdev = 37.067/38.516/40.959/1.734 ms
```

Revisa el manual del comando `ping` y responde a las siguientes preguntas.

25. ¿Qué tipo de mensaje es un ping? ¿Qué protocolos utiliza?

26. ¿Qué hace la opción `-f`? ¿Por qué para usarla hay que ser superusuario?

27. Haz un ping a todos los ordenadores del laboratorio usando *broadcast*

5.5. Herramienta `traceroute`

La herramienta `traceroute` (`tracert` en MS-DOS y derivados) sirve para descubrir el camino que siguen los paquetes desde el propio equipo hasta el destino especificado. Ejemplo:

```
lab102-194:~$ traceroute www.google.com
traceroute to www.google.com (74.125.39.105), 30 hops max, 60 byte packets
1  155.210.154.254 (155.210.154.254)  0.984 ms  1.272 ms  1.488 ms
2  155.210.251.9 (155.210.251.9)  0.281 ms  0.329 ms  0.388 ms
3  155.210.248.41 (155.210.248.41)  0.598 ms  0.801 ms  0.888 ms
4  155.210.248.66 (155.210.248.66)  1.763 ms  2.073 ms  2.099 ms
5  193.144.0.169 (193.144.0.169)  1.687 ms  1.963 ms  1.961 ms
6  GE0-2-0.EB-Zaragoza0.red.rediris.es (130.206.195.13)  2.123 ms  2.025 ms  2.983 ms
7  XE1-1-2.ciemat.rt1.mad.red.rediris.es (130.206.245.5)  10.727 ms  10.469 ms  10.281 ms
8  mad-b1-link.telia.net (213.248.81.25)  11.827 ms  12.053 ms  12.001 ms
9  prs-bb1-link.telia.net (213.155.131.152)  39.742 ms prs-bb1-link.telia.net (80.91.245.58)  40.045 ms
prs-bb2-link.telia.net (80.91.245.60)  34.814 ms
10  ffm-bb2-link.telia.net (80.91.246.184)  43.939 ms ffm-bb2-link.telia.net (80.91.246.180)  43.901 ms
ffm-bb2-link.telia.net (80.91.246.182)  43.910 ms
11  s-bb1-link.telia.net (80.91.246.211)  76.116 ms s-bb2-link.telia.net (80.91.251.146)  65.403 ms
s-bb2-link.telia.net (80.91.248.58)  120.067 ms
12  s-b3-link.telia.net (80.91.253.226)  65.712 ms s-b3-link.telia.net (80.91.249.220)  75.565 ms
s-b3-link.telia.net (213.155.131.121)  75.565 ms
13  google-ic-130574-s-b3.c.telia.net (213.248.93.194)  66.620 ms  66.954 ms  72.173 ms
14  209.85.250.192 (209.85.250.192)  84.284 ms  66.893 ms  74.520 ms
```

```
15  * * *
16  * * *
17  209.85.254.114 (209.85.254.114)  73.455 ms  73.464 ms 209.85.254.116 (209.85.254.116)  73.542 ms
18  * 209.85.249.166 (209.85.249.166)  76.278 ms *
19  fx-in-f105.1e100.net (74.125.39.105)  72.922 ms  68.083 ms  68.045 ms
```

Revisa el manual del comando `traceroute` y responde a las siguientes preguntas.

28. ¿Qué campo del protocolo IP usa este programa para que los paquetes no lleguen más allá de cierta distancia?
29. Si todo va bien, ¿cómo sabe hasta qué máquina ha llegado cada paquete?
30. ¿Qué indican los asteriscos que aparecen a veces?
31. ¿Cuántos paquetes sonda se envían a cada distancia específica?
32. Observa que, para ciertas distancias, las respuestas vienen de direcciones distintas. ¿Qué puedes deducir en esos casos?

En Internet existen muchas páginas que proporcionan herramientas de red (ping, traceroute, etc.). Por ejemplo, `http://geotraceroute.com` permite visualizar sobre un mapa las rutas seguidas desde una lista de posibles orígenes hasta el destino indicado. Visita esa web y en *Preferences*, en *Geek mode*, selecciona *Enabled* para que muestre todos los nodos atravesados de los que obtenga información.

33. Desde esa web, haz un traceroute desde *US - Los Angeles* hasta la direccción IPv4 de tu equipo. ¿Qué dos ciudades conecta el cable transoceánico que se atraviesa?
34. Haz un traceroute desde *BR - Rio de Janeiro* hasta la direccción IPv4 de tu equipo. Ten en cuenta que tu equipo pertenece a una red de investigación/educación y que hay un enlace transoceánico entre las redes de investiación y educación RedCLARA (que incluye a Brasil) y GÉANT (que incluye a España). ¿La ruta usada pasa por ese enlace transoceánico?
35. Haz ahora un traceroute desde *BR - Rio de Janeiro* hasta *xunta.es* (que no pertenece a RedIRIS). ¿Por qué la ruta ahora es tan distinta a la anterior?

5.6. ¿Sabías que...?

- Desde hace años hay planes para crear un punto neutro en Aragón (*Aragonix*). En concreto, el I Plan Director de Infraestructuras de Telecomunicaciones de Aragón[1], presentado en 2006, contemplaba la creación de un punto neutro de Aragón para conectar «tanto los proveedores aragoneses de acceso a Internet, la Universidad y centros de investigación, la Administración y los proveedores de acceso a Internet de carácter nacional». Posteriormente, en 2009, el Plan Director 2.0 para el Desarrollo de la Sociedad de la Información en la Comunidad Autónoma de Aragón[2] incluía entre sus acciones «la implementación de un punto neutro que permitirá la interconexión de todas las Administraciones públicas aragonesas entre sí y con la red SARA[3]».

[1] `http://www.aragon.es/estaticos/ImportFiles/24/docs/Areas/SocInfo/PlnEstrtgAra/PlanDir/I_PLAN_DIRECTOR_INFRAESTRUCTURAS_TELECOMUNICACIONES_ARAGON.pdf`

[2] `http://www.aragon.es/estaticos/ImportFiles/24/docs/Areas/SocInfo/PlnEstrtgAra/IIPlan/II_PLAN_DIRECTOR_SOCIEDAD_INFORMACION_ARAGON.pdf`

[3] La red SARA es la infraestructura común de comunicaciones del Estado español.

Práctica 6

Herramientas básicas de red

6.1. Objetivos

Uso de herramientas y comandos para visualizar y configurar conexiones a Internet.

6.2. Introducción

Para conectarse a Internet, un equipo necesita los siguientes datos: dirección IP de su interfaz de red (podría tener más de uno), dirección IP de su encaminador por defecto y dirección IP de su servidor de nombres de dominio (DNS). Esta información la puede obtener automáticamente a través del protocolo Dynamic Host Configuration Protocol (DHCP) si en la red hay un servidor DHCP activo. De esta forma, al arrancar, el ordenador solicita dicha información y el servidor se la proporciona. Actualmente esto es lo habitual en redes domésticas y en redes inalámbricas. En otros entornos es necesario configurar manualmente estos parámetros. A continuación se describe el uso de algunas herramientas para consultar y modificar la configuración de red y para comprobar su correcto funcionamiento.

6.3. Interfaces de red

Un interfaz de red es el punto de interconexión entre un equipo y una red. El interfaz de red más habitual es una tarjeta de red (*network interface card*, NIC). Existen también interfaces de red que no son dispositivos hardware, por ejemplo, el interfaz local (*loopback interface*).

Un sistema conectado a Internet necesita una dirección IP asociada al interfaz con dicha red. En GNU/Linux, la consulta o modificación de la configuración de interfaces se realiza mediante el comando `ifconfig`, mientras que en Windows se usa `ipconfig`. Para interfaces inalámbricos, `iwconfig` permite visualizar y configurar sus características específicas. Las distribuciones modernas de GNU/Linux están en proceso de sustituir los comandos de configuración originarios de BSD por el comando `ip`. Este comando engloba los anteriores y además permite configurar elementos como multicast, túneles, control de tráfico, IPsec, etc. Puedes encontrar ejemplos de uso de `ip` en Internet[1]. En general, cada sistema puede tener un comando específico, aunque su funcionamiento suele ser similar.

Conéctate por `ssh` a `lab000.cps.unizar.es` y ejecuta el comando `ifconfig` (situado en `/sbin/`). Cada interfaz (a la izquierda) muestra detalles sobre su configuración, incluyendo direcciones, datos transmitidos (TX) y datos recibidos (RX) hasta el momento. En un equipo de usuario, los interfaces tienen nombres como `eno0` o `enp2s0` para Ethernet (anteriormente `eth0`[2]), `wls1` para wlan (anteriormente `wlan1`), o tener etiquetas dependientes de la tarjeta de red.

1. A partir de la configuración, ¿qué interfaz tiene una dirección IPv4 asociada? ¿Cuál es su MTU?

[1] `http://dougvitale.wordpress.com/2011/12/21/deprecated-linux-networking-commands-and-their-replacements/`

[2] Desde la versión v197 de `systemd/udev` se asignan nombres predecibles y persistentes a los interfaces de red. Más información en `https://www.freedesktop.org/wiki/Software/systemd/PredictableNetworkInterfaceNames/`

2. ¿Qué es el interfaz `lo`? ¿Cuál es su MTU? ¿Qué direcciones IPv4 e IPv6 tiene asignadas? ¿Por qué no tiene asociada una dirección hardware (MAC) como otros interfaces?

3. Ejecuta el comando `ip addr` y comprueba que esencialmente aparece la misma información.

4. ¿Cuál de los dos comandos muestra el tiempo de validez de las direcciones IP asignadas? Observa que se va decrementando con el tiempo.

Los equipos del laboratorio 1.02, además, tienen configurados puentes (*bridges*) virtuales (`br0`, `virbr0`). Estos conmutadores virtuales se configuran con el comando `brctl` y permiten conectar máquinas virtuales de forma equivalente a como se conectarían si fueran máquinas reales. A partir de ahora vamos a trabajar en la máquina local (lab102-yyy, siendo yyy un número entre 191 y 210).

5. Ejecuta `ifconfig` y observa la salida. ¿Cuántas tarjetas de red hay?

6. Ejecuta `brctl show br0` y observa la salida. ¿A qué interfaz de red está conectado el puerto `br0`?

7. Ejecuta `brctl showmacs br0` y observa la salida. ¿Qué información se está mostrando?

8. Ejecuta `brctl showstp br0` y observa la salida. ¿A qué te suenan los parámetros mostrados?

6.4. Conectividad local

Si estuviéramos en una red aislada (sin encaminador hacia Internet) tendríamos conectividad local, es decir, podríamos comunicarnos con el resto de equipos conectados a esa misma red local. Para ello, el ordenador construirá paquetes cuya dirección IP destino conoce (es con quien se quiere comunicar), pero posiblemente no conozca su identificador MAC, que debe especificar como destino en la cabecera ethernet (o el protocolo de interfaz de red que corresponda), así que necesita algún medio para obtener esa información. En clase hemos visto que en IPv4 las direcciones lógicas (IP) se asocian a los identificadores físicos (MAC) mediante el protocolo ARP (*Address Resolution Protocol*), mientras que en IPv6 esas asociaciones se gestionan a través de ICMPv6 (*Neighbour Discovery*). Es decir, cualquier equipo tiene una tabla con asociaciones entre direcciones IP y sus correspondientes identificadores MAC. Así, cuando se necesita un identificador MAC se genera un mensaje ARP (o ICMPv6) de difusión total (*broadcast*) del tipo «Quien tenga esta dirección IP, que me diga su identificador MAC». Ese mensaje llegará a todos los equipos de la red local, y el equipo aludido responderá. Con esa información se actualiza la tabla de asociaciones IP-MAC. El comando `arp` permite ver y manipular esta tabla de asociaciones en IPv4, mientras que el comando `ip neigh` funciona tanto para IPv4 como para IPv6.

9. Comprueba qué asociaciones tienes en este momento.

10. Haz un `ping` a algún equipo del laboratorio cuya dirección no esté entre las asociaciones anteriores y posteriormente vuelve a comprobar la tabla de vecinos. ¿Qué información nueva se ha añadido?

6.5. Tablas de reexpedición/encaminamiento

Una función importante del nivel de red es la reexpedición del tráfico. La tabla de reexpedición/encaminamiento especifica hacia dónde hay que mandar un paquete dependiendo de su dirección destino.

6.5.1. IPv4

El comando `route` (actualizado por el comando `ip route`) permite consultar las rutas actuales y configurar las rutas de forma estática, es decir, manualmente sin que intervenga ningún protocolo de búsqueda de caminos óptimos. Lanzado sin argumentos, el comando `route` muestra la tabla de reexpedición/encaminamiento de tu equipo. Observa también la información mostrada por el comando `ip route`.

11. ¿Cuál es el encaminador por defecto de tu equipo?

12. ¿A qué redes está conectado tu equipo? ¿Cuál es la máscara de cada una de ellas en formato CIDR? ¿Se corresponden los bits a los mostrados en *Genmask*?

13. ¿Qué indica el que para ciertos destino no haya *vía* para llegar a ellos?

14. ¿Cuál sería el comando para añadir una ruta a la red 145.145.20.0 con máscara 255.255.255.0, pasando por el encaminador (*gateway*/siguiente salto) 145.145.20.1 a través del interfaz eth2?

15. ¿Qué indica *Metric*?

En GNU/Linux, uno de los métodos de comunicación entre el núcleo y el resto del sistema es a través del sistema de ficheros virtual montado en `/proc`, donde cada fichero es una especie de variable con cierto valor (o valores). Por ejemplo, el contenido del fichero `/proc/sys/net/ipv4/ip_forward` indica si el equipo está actuando como encaminador (1) o no (0). Si el equipo no actúa como encaminador, cuando recibe un paquete con una dirección IP que no le pertenece, directamente lo descarta. En cambio, si está actuando como encaminador, reexpedirá ese paquete como corresponda según su tabla.

16. ¿Está actuando tu equipo como encaminador?

Por otro lado, en la asignatura hemos visto muchos algoritmos que dependen de parámetros. En la mayoría de los casos estos parámetros se pueden cambiar, dependiendo del sistema operativo. En GNU/-Linux se puede interactuar con el sistema mediante la función *setsockopt()* (específica para parámetros de red), el directorio `/proc/sys/` o el comando `sysctl` (en `/sbin`). Este comando proporciona y permite modificar la misma información que hay en el directorio `/proc/sys/`. Lanza `sysctl -a` para mostrar todos los parámetros (puedes filtrar los relativos a las redes con `|grep net`). También puedes buscar el valor de un parámetro si conoces el nombre que tiene.

17. ¿Coincide el valor de `net.ipv4.ip_forward` con el de la pregunta anterior?

18. ¿En qué se diferencia la tabla de un ordenador normal de la de un encaminador?

6.5.2. IPv6

La configuración usando IPv6 es muy similar a la de IPv4. La diferencia más importante es que en IPv6 existe la autoconfiguración *sin estado*, en la que los equipos rellenan automáticamente los campos de sus direcciones IP con la información que conocen. De esta forma, al arrancar un equipo se autoconfigura como mínimo con una dirección localmente válida.

Entra en `http://test-ipv6.com/` para comprobar tu conectividad IPv6 y observa los resultados. Revisa las *Pruebas ejecutadas* y la *Información técnica*.

19. ¿Qué puedes deducir?

Revisa las direcciones IPv6 en las transparencias de clase. Busca en el manual de `route` cómo mostrar las rutas del protocolo IPv6 y muéstralas. Haz lo mismo para el nuevo comando `ip route`.

20. ¿Hay alguna red con dirección globalmente única en los destinos? ¿Hay encaminador por defecto? ¿Qué implica eso?

21. ¿Qué simboliza el prefijo `fe80`? ¿Qué implica lo que aparece en sus campos *Next Hop*?

22. ¿Qué simboliza el prefijo `ff`?

6.6. Servidores de nombres de dominio

Con todo lo anterior correctamente configurado, el equipo ya tiene conectividad a Internet. Aún así, existe un servicio adicional que se considera básico. Ese servicio es el servidor de nombres, que realiza traducciones de nombres *fácilmente usables por personas* a direcciones IP. Los nombres o *dominios* son jerárquicos. Por ejemplo, todos los nombres dentro del dominio de España acaban en *.es* y todos los equipos de la Universidad de Zaragoza acaban en *.unizar.es*. En general, como mínimo cada red dispone de dos equipos encargados de la traducción de nombres. Para que nuestro equipo los conozca, hay que especificar cómo llegar a ellos, es decir, su dirección IP. En GNU/Linux, esa especificación se encuentra en el fichero `/etc/resolv.conf`, que también incluye el dominio que añadirá por defecto a los nombres que vaya a traducir.

23. ¿Cuáles son los servidores de nombres de tu máquina?
24. ¿Por qué están especificados con su dirección IP y no con su nombre?
25. ¿Qué pasaría si fallaran los dos?
26. Si tienes una conexión TCP activa con *www.google.com*, ¿qué pasaría con esa conexión si todos los servidores de nombres fallaran?

Una forma sencilla de realizar consultas de nombres es a través del comando `host`. Revisa el manual del comando y observa la respuesta al preguntar por los siguientes nombres:

27. *hendrix*
28. *moodle* (¿por qué *hendrix* funciona y *moodle* no?)
29. *moodle.unizar.es*
30. *hendrix.* (no te dejes el . final)
31. *moodle.unizar.es.* (no te dejes el . final) (¿por qué *moodle.unizar.es.* funciona y *hendrix.* no?)
32. *unizar.es.* (no te dejes el . final)
33. *unizar.es* (¿por qué *unizar.es* funciona con y sin . final?)
34. *www.unizar.es.*
35. Además de traducción de direcciones, ¿qué otras dos informaciones te han aparecido en algunas de las consultas anteriores?

El comando `dig` también permite realizar consultas de nombres de forma similar a `host`, pero proporcionando muchos más detalles. Los principales registros que DNS maneja son:

- A (Address), define la dirección IPv4
- AAAA (Address), define la dirección IPv6
- NS (Name Server), define los servidores DNS
- MX (Mail eXchanger), define los servidores de correo
- CNAME (Canonical Name), permite definir alias de otros nombres
- SOA (Start Of Authority), contiene información sobre el servidor DNS primario
- LOC (LOCation), define la localización
- TXT (TeXT): almacena cualquier información

36. Ejecuta `dig ANY unizar.es` y compara el resultado con la información obtenida con `host`.

En cuanto a los servidores de nombres de dominio, el uso de IPv6 simplemente implica manejar un nuevo *tipo* de información: las direcciones IPv6. Si las direcciones de IPv4 (32 bits) se simbolizan con tipo A (*address*), las direcciones IPv6 (4 veces mayores) se simbolizan con tipo AAAA.

37. Pregunta por tipos AAAA en *google* (`dig AAAA google.com`). ¿Responde con una dirección IPv6?

Obtén la dirección IP de los siguientes nombres:

38. *ipv6.google.com.*

39. *www.v6.facebook.com.*

40. ¿Qué obtienes si haces `ping` de *ipv6.google.com*? ¿Por qué?

41. ¿Y si haces `ping6`? ¿Por qué?

6.7. Estado de puertos

La herramienta `netstat` permite visualizar múltiples datos de red del ordenador en el que nos encontramos, incluyendo información de rutas e interfaces. No obstante, se usa particularmente para mostrar el estado de los puertos TCP y UDP. Al igual que otros comandos que has visto en la asignatura, esta herramienta está siendo sustituida, en este caso por el comando `ss` (*socket statistics*) en GNU/Linux.

42. ¿Qué hace `netstat` con el argumento `-t`? ¿Qué muestran las columnas *Local Address* y *Foreign Address*?

43. Prueba ahora `ss -t` ¿Se parece?

44. ¿Qué hace el argumento `-l`? ¿Por qué se muestran asteriscos en la columna *Foreign Address* (*Peer Address* en `ss`)?

45. ¿Qué hace el argumento `-a`? Observa que además de sockets TCP/UDP aparecen también los *UNIX domain sockets* asociados a un fichero, que habrás estudiado en Sistemas Operativos.

46. Lanza un `netcat` como servidor TCP y en otro terminal obtén un listado de los sockets TCP que estén en modo *listen* (e.g. `ss -l -t`). ¿Puedes localizar el socket del `netcat` en el listado? ¿Qué aparece en la columna *state*?

47. Lanza ahora un `netcat` que se conecte con el `netcat` anterior. Localiza su entrada con netstat o ss (e.g. `ss -t -a`). ¿Cuántas veces aparece? ¿En qué estado está ahora el socket? ¿Qué puertos se están utilizando en esa conexión?

48. Si pulsas Ctrl+C en uno de los `netcat`, ¿finaliza la conexión de ese `netcat` o de los dos? ¿Por qué?

49. Si inmediatamente después de cerrar la conexión ejecutas `ss -a -t` ¿en qué estado aparece la conexión?

50. Lanza ahora un `netcat` como servidor UDP y en otro terminal obtén un listado de los sockets UDP (`ss -u -l`). ¿Puedes localizar el socket del `netcat` en el listado?

51. Lanza ahora un `netcat` UDP para interactuar con el `netcat` anterior, pero sin enviar ningún texto entre ellos. ¿Cuántas entradas aparecen en el listado referidas a los sockets utilizados (`ss -u -l -a`)? ¿Cuál es su estado?

52. Escribe algo en el `netcat` *servidor*. ¿Se transmite al cliente? ¿Por qué? ¿Ha cambiado la información que muestra `ss`?

53. Escribe algo *distinto* ahora en el `netcat` *cliente*. ¿Qué ha pasado? ¿Ha cambiado la información que muestra `ss`?

54. Pulsa Ctrl+C para finalizar el servidor. ¿Ha finalizado el cliente automáticamente? ¿Cuántas entradas aparecen ahora en el listado? ¿En qué estado?

55. Sin cancelar el cliente anterior, lanza un nuevo servidor `netcat` UDP en el mismo puerto. ¿Si escribes algo en el cliente, lo recibe el nuevo servidor? ¿Por qué?

Al igual que con los parámetros de la capa de red anteriores, puedes mostrar también los valores de parámetros de capa de transporte mediante el comando `sysctl`.

56. ¿Qué valor tiene el factor de escalado (opcional) de la ventana anunciada del protocolo TCP (`sysctl net.ipv4.tcp_window_scaling`)?

57. ¿Qué valores (mínimo, por defecto y máximo) tiene la ventana de recepción del protocolo TCP (`sysctl net.ipv4.tcp_rmem`)?

58. ¿Qué valores (mínimo, por defecto y máximo) tiene la ventana de emisión del protocolo TCP (`sysctl net.ipv4.tcp_wmem`)?

6.8. Interacción con protocolos de aplicación

En prácticas anteriores se ha utilizado la herramienta `netcat` para observar y verificar el comportamiento de aplicaciones como la de enviar vocales. Esto es generalizable para cualquier protocolo de aplicación cuyas comunicaciones estén basadas en texto, por ejemplo el protocolo HTTP.

59. Lanza el `netcat` como servidor TCP en tu equipo, por ejemplo en el puerto 32002. A continuación introduce `http://pon-aquí-tu-direccion-ip:32002/` en un navegador (usando la dirección IP de tu equipo.) ¿Qué mensaje ha recibido `netcat`?

60. Usa ahora netcat como cliente para realizar una petición web (`nc -C wwww.unizar.es 80`). Escribe exactamente lo siguiente, respetando mayúsculas y minúsculas, y sin olvidar la línea en blanco final:
```
GET / HTTP/1.1
Host: www.unizar.es

```
Observa el mensaje recibido y explica por qué es diferente al de la pregunta anterior.

6.9. Herramienta Nmap

Otra herramienta muy interesante es `nmap`, que permite explorar y analizar muchos aspectos de las redes. Muchas exploraciones las realiza usando los sockets de forma «normal». En cambio, para hacer ciertas exploraciones menos convencionales, esta herramienta construye «manualmente» los paquetes, es decir, rellenando los campos de las cabeceras con ciertos valores (correctos o no). En general los sistemas no permiten que cualquier usuario pueda hacer esto, así que para realizar ciertas exploraciones es necesario tener permisos de administrador. Eso sí, ten en cuenta que cualquier exploración implica que la máquina explorada debe responder, con lo que además de saber que está siendo explorada conoce la dirección de quien la está explorando.

61. Revisa el manual del comando `nmap` y haz una exploración de la red del laboratorio mediante `ping`.

6.10. ¿Sabías que...?

- Aunque la mayoría de servicios y contenidos alojados en «Internet IPv6» también lo están en «Internet IPv4», hay algunos (cada vez más) que sólo están disponibles usando IPv6. Ciertas empresas proporcionan servicios gratuitos de túnel IPv6 sobre IPv4, para acceder a «Internet IPv6» desde proveedores de servicios que sólo proporcionan IPv4. Por ejemplo puedes configurar un túnel de estas características en `http://www.tunnelbroker.net/`
- La herramienta `nmap` es una de las preferidas por los hackers en el cine desde su aparición en *Matrix Reloaded* (`http://nmap.org/movies.html`).