Aprendiendo a programar con `Libpcap`

Alejandro López Monge
kodemonk@emasterminds.net

20 de Febrero de 2005

`Libpcap` es una de esas herramientas que por su flexibilidad y potencia merece ser introducida en el grupo de las [META | COJO] herramientas. Si los `RAW` sockets son nuestra voz en la red, `Libpcap` será nuestros oidos.

Índice

1. Introducción

Antes de comenzar con `Libpcap` es fundamental que comprendamos las bases del funcionamiento de una Red Local. No pretendo dar una definición exhaustiva ni formal de cómo funciona una red, lo importante ahora es que seas capaz de identificar cada uno de los elementos que la componen.

1.1. Introducción a LANs Ethernet

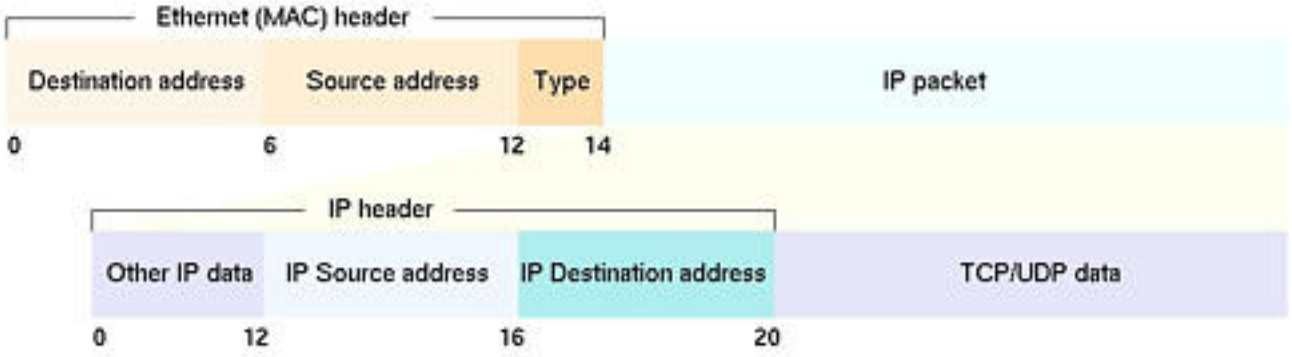
Qué mejor manera para repasar cada uno de los elementos que componen una red local que seguir el recorrido de uno de sus muchos paquetes.

Cuando una aplicación necesita enviar información a traves de la red, lo primero que hace es encapsular esa información en un protocolo de transporte (TCP,UDP), a partir de ese momento la información pasa a llamarse `payload` o carga útil. Un protocolo de transporte sirve para especificar cómo queremos que viaje nuestro paquete. Si lo pensamos esto no es nada nuevo, cuando nos vamos de vacaciones a Benidor enviamos postales a la familia, pero cuando tenemos que enviar un expediente académico lo hacemos por correo certificado. La razón para elegir un método u otro depende de lo importante que sea la información, del coste, de la prisa que tengamos etc... Este concepto es perfectamente aplicable a las redes, ya que cada protocolo de transporte tiene unas ventajas y unos inconvenientes.

Ya sabemos que nuestro paquete va a ir por correo certificado, pero ahora... ¿cómo especificamos el destinatario?. Todas las máquinas de una misma red tienen asociado un identificador único en la red. Este identificador se llama dirección IP y es un direccionamiento lógico, es decir, independiente de la arquitectura sobre la que esté montada la red (Ethernet,Token Ring,Token Bus etc...).

Una vez añadida la cabecera IP a nuestro paquete, sólo nos faltan por añadir las cabeceras propias del tipo de red sobre el que va a moverse el paquete, en nuestro caso la cabecera Ethernet. (Ver Fig.1)

Fig.1 Encapsulado de datos

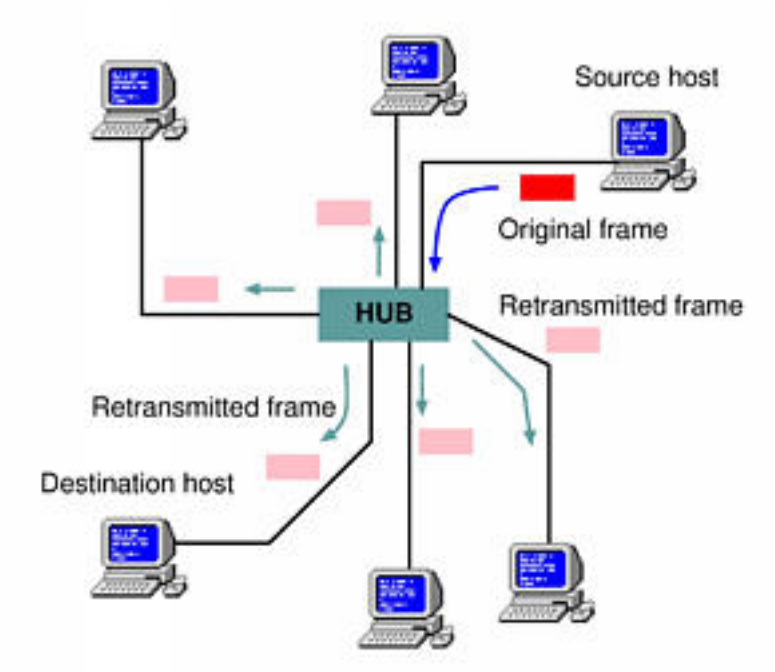


En una LAN Ethernet todo dispositivo de red direccionable tiene asociada una dirección Ethernet (normalmente llamada dirección MAC o simplemente MAC) que viene grabada de serie por el fabricante. Una de las cosas con las que se informa la cabecera Ethernet es la MAC de cada extremo de la comunicación.

¿Cómo sabe la red cual es la dirección MAC asociada a una IP?, la respuesta más sencilla seria dotar a cada máquina de una relación de equivalencias IP-MAC. Esta solución existe en sistemas reales y tiene el inconveniente de ser muy constoso de mantener y poco flexible. De ahí que las redes Ethernet cuenten con un mecanismo de resolución de direcciones IP a MAC y viceversa, estos protocolos se llaman `ARP` y `RARP`.

Una vez que tenemos unida toda esa información `Payload + TCP + IP + Ethernet` lo que en su conjunto se llama `frame Ethernet` podemos comenzar la transmisión. En un modelo sencillo como en el que estamos trabajando, el paquete no se dirige directamente a su destinatario sino que es copiado por el Hubb en todos los cables de Red (Ver Fig. 2). Todos los sistemas comenzarán a leer los primeros 6 bytes del frame en los que está codificada la MAC destino, pero sólo aquel cuya MAC coincida con la del destinatario, leerá el resto del paquete. Cómo puede verse, la red no tiene ningún mecanismo para asegurar que un envio sólo pueda ser leido por su legítimo destinatario, basta un pequeño cambio de configuración en el modo en que el driver de red captura los datos (configuración en *modo promiscuo*), para que nuestra máquina reciba todos los paquetes que circulan por la red.

Fig.2 Envio de un frame Ethernet en una LAN



2. Libpcap

2.1. Libpcap, un poco de historia

`Libpcap` es una libreria *open source* escrita en C que ofrece al programador una interfaz desde la que capturar paquetes en la capa de red, además `Libpcap` es perfectamente portable entre un gran número de S.O's.

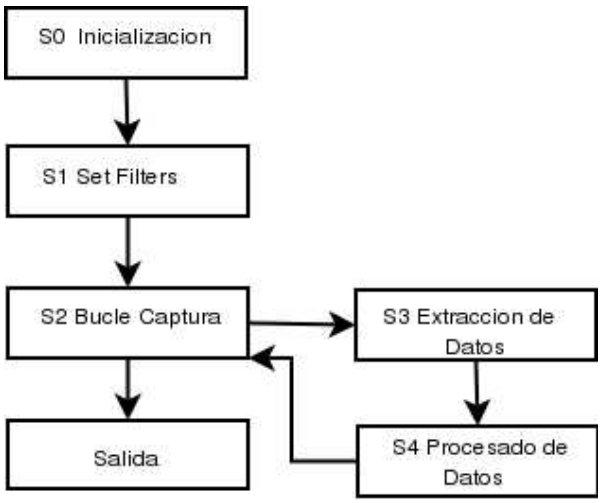
2.2. Primeros pasos

Antes de comenzar a programar con `Libpcap`, debes tener instalada una copia en tu máquina, las fuentes oficiales de libpcap están en `www.tcpdump.org`. La instalación es rápida e indolora, basta con seguir los tres pasos habituales: configurar(./configure), construir (make) e instalar como root (make install).

2.3. Esquematización de un programa

No importa lo complicado que sea el programa que vayamos a construir con Libpcap, este siempre tiene que seguirá un esquema básico (Ver Fig.3)

Fig.3 Esquematización de un programa con Libpcap



En las siguientes secciones se desarrollan con ejemplos cada uno de las fases mostradas en la Fig.3 numeradas como S1, S2, S3...

2.4. (S0) Obteniendo información del sistema

Como puede verse en el gráfico, la primera fase de nuestro programa es la Inicialización. Ésta engloba las funciones capaces de obtener información del sistema: Las interfaces de red instaladas, los configuración de estas interfaces (Máscara de Red, Dirección de Red) etc... a continuación se recogen las funciones más relevantes.

2.4.1. Funciones específicas

`char *pcap_lookupdev(char *errbuf)`

Devuelve un puntero al primer dispositivo de red válido para ser abierto para captura, en caso de error se devuelve `NULL` y una descripción del error en `errbuf`.

`int pcap_lookupnet(char *device, bpf_u_int32 *netp,bpf_u_int32 *maskp, char *errbuf)`

Una vez obtenido el nombre de una interfaz válida, podemos consultar su dirección de red (que no su dirección IP) y su máscara de subred. `device` es un puntero a un array de caracteres que contiene el nombre de una interfaz de red válida, `netp` y `maskp` son dos punteros a `bpf_u_int32` en los que la función dejará la dirección de red y la mascara respectivamente. En caso de error la función devuelve -1 y una descripción del error en `errbuf`.

`int pcap_findalldevs(pcap_if_t **alldevsp, char *errbuf)`

Está función nos devuelve todos las interfaces de red que pueden ser abiertas para capturar datos, puede que existan más interfaces en el sistema pero que por una razón u otra no puedan ser abiertas para captura (falta de permisos). Estas interfaces no aparecerán en la lista.
Para llamar a esta función nos basta un puntero sin inicializar de tipo `pcap_if_t`, vease las sección *Estructuras de datos usadas por Pcap* página 39 para ver una descripción detallada. La función transformará ese puntero en una lista enlazada que contendrá cada una de las interfaces y sus datos asociados (dirección de red y mascara). En caso de error la función devuelve -1 y una descripción del error en errbuf.

`int pcap_datalink(pcap_t *p)`

Esta función devuelve el tipo de enlace de datos asociado a una interfaz de red. El valor de retorno puede ser uno de los siguientes:

DLT_NULL BSD loopback encapsulation

DLT_EN10MB Ethernet (10Mb, 100Mb, 1000Mb, and up)

DLT_IEEE802 IEEE 802.5 Token Ring

DLT_ARCNET ARCNET

DLT_SLIP SLIP

DLT_PPP PPP

DLT_FDDI FDDI

DLT_ATM_RFC1483 RFC 1483 LLCSNAP-encapsulated ATM

DLT_RAW raw IP, el paquete comienza con una cabecera IP

DLT_PPP_SERIAL PPP en modo HDLC-like framing (RFC 1662)

DLT_PPP_ETHER PPPoE

DLT_C_HDLC Cisco PPP con HDLC framing, definido en 4.3.1 RFC 1547

DLT_IEEE802_11 IEEE 802.11 wireless LAN

DLT_LOOP OpenBSD loopback encapsulation

DLT_LINUX_SLL Linux `cooked` capture encapsulation

LT_LTALK Apple LocalTal

2.4.2. Ejemplo Primer programa con Libpcap

```
/************************************************************************
*
* Fichero: lsdevs.c
* Fecha:  Alejandro Lopez Monge
* Orignal: Martin Casado http://www.cet.nau.edu/~mc8/Socket/Tutorials/section1.html
*
* Compilacion: gcc lsdevs.c -lpcap
*
* Descripcion:
*  Buscamos la primera interfaz de red disponible y lista su direccion de red
*  (que no su direccion IP) y su mascara de subred.
*
************************************************************************/

#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <pcap.h>                       //include a libpcap

int main(int argc, char **argv)
{
  char *net;                            // direccion de red
  char *mask;                           // mascara de subred
  char *dev;                            // nombre del dispositivo de red
  int ret;                              // codigo de retorno
  char errbuf[PCAP_ERRBUF_SIZE];        // buffer para mensajes de error
  bpf_u_int32 netp;                     // direcion de red en modo raw
  bpf_u_int32 maskp;                    // mascara de red en modo raw
  struct in_addr addr;

  if((dev = pcap_lookupdev(errbuf))== NULL) //conseguimos la primera interfaz libre
  {printf("ERROR %s\n",errbuf);exit(-1);}

  printf("Nombre del dispositivo: %s\n",dev); //mostramos el nombre del dispositivo

  if((ret = pcap_lookupnet(dev,&netp,&maskp,errbuf))==-1) //consultamos las direccion de red y las mascara
  {printf("ERROR %s\n",errbuf);exit(-1);}

  addr.s_addr = netp;                   //Traducimos la direccion de red a algo legible
  if((net = inet_ntoa(addr))==NULL)
  {perror("inet_ntoa");exit(-1);}

  printf("Direccion de Red: %s\n",net);

  addr.s_addr = maskp;                  //Idem para la mascara de subred
  mask = inet_ntoa(addr);

  if((net=inet_ntoa(addr))==NULL)
  {perror("inet_ntoa");exit(-1);}

  printf("Mascara de Red: %s\n",mask);
  return 0;
}
```

2.5. (S2) Capturando Paquetes

Una vez que sabemos cómo obtener el listado de las interfaces instaladas en nuestro sistema y sus configuraciones, ya estamos preparados para comenzar con la captura en sí.

Existen varias funciones para capturar paquetes, las prinicipales diferencias entre ellas son: El número de paquetes que queremos capturar, el modo de captura, normal o promiscuo y la manera en que se definen sus funciones de llamada o Callbacks (la función invocada cada vez que se captura un paquete).

2.5.1. Funciones Específicas

`pcap_t *pcap_open_live(char *device, int snaplen,int promisc, int to_ms, char *errbuf)`

Antes de entrar en el bucle de captura hay que obtener un descriptor de tipo `pcap_t`, para lo cual empleamos esta función. El primer parametro (`char* device`) es el nombre del dispositivo de red en el que queremos iniciar la captura (los valores `ANY` o `NULL` fuerzan la captura en todos los dispositivos disponibles).

El segundo argumento (`int snaplen`) especifica el número máximo de bytes a capturar. El argumento promistic indica el modo de apertura, un valor distinto de 0 iniciara la captura en modo promiscuo, 0 para modo normal.

Cómo se explicará más adelante en detalle en la sección **2.6.1**, los programas basados en `Libpcap` se ejecutan en la zona de usuario pero la captura en sí en la zona de Kernel. Se hace por lo tanto necesario un cambio de área, estos cambios son muy costosos y hay que evitarlos a toda costa si queremos optimizar el rendimiento, de ahí que esta función nos permita especificar mediante el parámetro `to_ms` cuantos milisegundos queremos que el Kernel agrupe paquetes antes de pasarlos todos de una vez a la zona de usuario.
Si la función devuelve NULL se habrá producido un error y puede encontrarse una descripción del error en `errbuf`.

`int pcap_dispatch(pcap_t *p, int cnt,pcap_handler callback, u_char *user)`

Esta función se utiliza para capturar y procesar los paquetes. `cnt` indica el número máximo de paquetes a procesar antes de salir, `cnt`=-1 para capturar indefinidamente.
`callback` es un puntero a la función que será invocada para procesar el paquete. Para que un puntero a función sea de tipo `pcap_handler`, debe recibir tres parámetros:

Puntero `u_char` Aquí es donde va propiamente paquete, más adelante veremos cómo interpretarlo.

Estructura `pcap_pkthdr` Definida en detalle en la sección de Estructuras página 39.

La función devuelve el número de paquetes capturados o -1 en caso de error, en cuyo caso pueden emplearse las funciones pcap_perror() y pcap_geterr() para mostrar un mensaje más descriptivo del error.

`int pcap_loop(pcap_t *p, int cnt, pcap_handler callback, u_char *user)`

Es bastante parecida a `pcap_dispatch`, la diferencia es que no finaliza cuando se produce un error por *timeout*. En caso de error se devolverá un número negativo y 0 si el número de paquetes especificados por cnt se ha completado con éxito.

`u_char *pcap_next(pcap_t *p, struct pcap_pkthdr *h)`

Lee un único paquete y devuelve un puntero a u_char con su contenido. Sin necesidad de declarar ninguna función Callback.

2.5.2. Ejemplo 1 (pcap_next)

```
/*********************************************************************
* file:     pcap_next.c
* date:   25-Abril-2005
* Author: Alejandro Lopez Monge
*
* Compilacion: gcc -lpcap -o pcap_next pcap_next.c
*
* Ejemplo de como capturar un unico paquete usando pcap_next
*********************************************************************/
                                                                      10
#include <stdio.h>
#include <stdlib.h>
#include <pcap.h>
#include <errno.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <netinet/if_ether.h>
int main(int argc, char **argv)
{                                                                     20
    int i;
    char *dev;
    char errbuf[PCAP_ERRBUF_SIZE];
    pcap_t* descr;
    const u_char *packet;
    struct pcap_pkthdr hdr;
    u_char *ptr;    //Contenedor del paquete

   if((dev = pcap_lookupdev(errbuf))==NULL) //buscamos un dispositivo
   {printf("%s\n",errbuf);exit(1);}                                    30
    printf("Abriendo: %s\n",dev);

   if((descr = pcap_open_live(dev,BUFSIZ,0,-1,errbuf)) == NULL) //abrimos un descriptor
              {printf("pcap_open_live(): %s\n",errbuf);exit(1);}

   if((packet = pcap_next(descr,&hdr))==NULL) //capturamos nuestro primer paquete
    {printf("Error al capturar el paquete\n");exit(-1);}

    printf("Capturado paquete de tamano %d\n",hdr.len);
    printf("Recibido a las %s\n",ctime((const time_t*)&hdr.ts.tv_sec));  40

  return 0;
}
```

2.5.3. Ejemplo 2 (pcap_loop)

```
/*************************************************************************
* file:     pcap_loop.c
* date:   25-Abril-2005
* Author: Alejandro Lopez Monge
*
* Compilacion: gcc -lpcap -o pcap_loop pcap_loop.c
*
* Ejemplo de como usar la funcion pcap_loop, y definicion de una
* funcion Callback
*                                                                        10
*************************************************************************/

#include <pcap.h>
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <netinet/if_ether.h>                                           20

//Funcion callback, sera invocada cada vez que se reciba un paquete
void my_callback(u_char *useless,const struct pcap_pkthdr* pkthdr,const u_char* packet)
{
    static int count = 1;
    fprintf(stdout,"%d, ",count);
    fflush(stdout);
    count++;
}
                                                                        30
int main(int argc,char **argv)
{
    char *dev;
    char errbuf[PCAP_ERRBUF_SIZE];
    pcap_t* descr;
    const u_char *packet;
    struct pcap_pkthdr hdr;
    struct ether_header *eptr; // Ethernet
    bpf_u_int32 maskp;         // mascara de subred
    bpf_u_int32 netp;          // direccion de red                     40


    if(argc != 2)
     {fprintf(stdout,"Modo de Uso %s \"programa de filtrado\"\n",argv[0]);return 0;}

    dev = pcap_lookupdev(errbuf); //Buscamos un dispositivo del que comenzar la captura
    if(dev == NULL)
     {fprintf(stderr,"%s\n",errbuf); exit(1);}
    else
     {printf("Abriendo %s en modo promiscuo\n",dev);}                    50

    pcap_lookupnet(dev,&netp,&maskp,errbuf); //extraemos la direccion de red y la mascara
```

```
    descr = pcap_open_live(dev,BUFSIZ,1,-1,errbuf); //comenzamos la captura en modo promiscuo

    if(descr == NULL)
     {printf("pcap_open_live(): %s\n",errbuf); exit(1); }

    pcap_loop(descr,-1,my_callback,NULL); //entramos en el bucle (infinito)
                                                                                    60
    return 0;
}
```

2.6. (S1) Cómo filtrar sólo lo que nos interesa

2.6.1. ¿Qué es un Packet/Socket Filter?

Ya hemos comentado que Libpcap es una libreria de funciones y que los procesos que ejecutan estas funciones, lo hacen en el nivel de usuario (*user space*). Sin embargo la captura real de datos tiene lugar en las capas más bajas del Sistema operativo, en la llamada zona Kernel (*Kernel area*), debe por lo tanto existir un mecanismo capaz de traspasar esta frontera, y hacerlo además de un modo eficiente y seguro, ya que cualquier cualquier fallo en capas tan profundas degradará el rendimiento de todo el sistema.

Supongamos que nos interesa programar una aplicación capaz de monitorizar la red en tiempo real en busca de paquetes cuyo puerto TCP origen sea el 135 (síntoma de que alguno de los ordendores de la red esta infectado por el Blaster). Si no existiese ningún mecanismo de filtrado, el Kernel no sabría cuales son los paquetes en los que esta interesada nuestra aplicación, por lo que tendría que traspasar la frontera `Kernel - User space` por cada paquete que transite por la red.
Para evitar esto, la solución pasa por que la aplicación establecezca un filtro en la zona Kernel que sólo deje pasar los paquetes cuyo puerto TCP destino sea el 135. Esta es la principal labor de un Packet/Socket Filter.

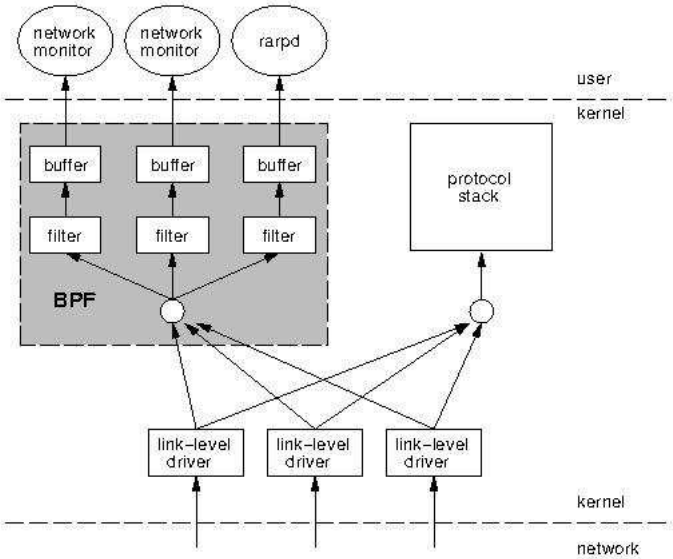
No existe un sistema único de filtrado, por el contrario prácticamente cada S.O reescribe su propia solución: `NIT` para SunOS, `Ultrix Packet Filter` para DEC Ultrix, BPF para sistemas BSD (originalmente) y más recientemente `LSF` (*Linux Socket Filter*) la implementacion para Linux.

En este apartado me centraré únicamente en `BPF` por ser el más extendido y el modelo de referencia para la creación de `LSF`.

El funcionamiento de `BPF` se basa en dos grandes componentes: El *Network Tap* y el *Packet Filter*. El primero es el encargado de recopilar los paquetes desde el driver del dispositivo de red y entregarselos a aquellas aplicaciones a las que vaya destinado. El segundo en el encargado de decidir si el paquete debe ser aceptado (coincidencia direcciones Ethernet) y en caso afirmativo, cuanto de ese paquete debe ser entregado a la aplicación (no tendría sentido entregarle a la aplicación un paquete con cabeceras Ethernet).

En la figura 4 puede verse el esquema general de funcionamiento de `BPF` dentro del sistema. Cada vez que llega un paquete a la interfaz de red, lo normal es que el driver de red lo mande hacia la pila de protocolos (protocol stack) siempre y cuando no esté activo `BPF` en cuyo caso antes de enviarse a la pila, el paquete debe ser procesado por él.

Fig.4 BPF en acción



BPF será el encargado de comparar el paquete con cada uno de los filtros establecidos, pasando una copia de dicho paquete a cada uno de los buffers de las aplicaciones cuyo filtro se ajuste a los contenidos del paquete. En caso de que no exista ninguna coincidencia, se devuelve el paquete al driver, el cual actuará normalmente, es decir, si el paquete esta dirigido a la propia máquina se lo pasará a la pila de protocolos y en caso contrario lo descartará sin que el paquete haya salido en ningún momento de la zona Kernel.

Puede haber procesos interesados en consultar cada uno de los paquetes de la red, lo que echa por tierra todos los intentos por maximizar el rendimiento ya que tendríamos que traspasar la frontera entre la zona Kernel y el espacio de usuario por cada paquete recibido.
Para evitar esto BPF agrupa la información de varios paquetes para luego pasarlos todos de una vez.
Para mantener la secuencialidad en que se recibieron los paquetes, BPF añade una marca de tiempo, tamaño y offset a cada uno de los paquetes del grupo.

2.6.2. Primitivas de Filtrado TCPDUMP

BPF tiene su propio lenguaje para la programación de filtros, un lenguaje de bajo nivel parecido al ensamblador. De modo que Libpcap implementa un lenguaje mucho más amigable para definir sus filtros. Este lenguaje lógicamente debe ser compilado a BPF compatible antes de ser aplicado.

A continuación se detalla cómo programar filtros con el lenguaje de alto nivel desarrollado por Libpcap y popularizado por TCPDUMP que a día de hoy se ha convertido en el estandar para definir filtros de captura.

La expresión que se usa para definir el filtro tiene una serie de primitivas y tres posibles modificadores a las mismas. Esta expresión puede ser verdadera en cuyo caso el paquete se pasa a la zona de usuario o falsa en cuyo caso el paquete se descarta sin llegar a salir en ningún caso de la zona Kernel, tal y como hemos visto en la sección anterior.

Los 3 modificadores posibles son:

tipo Puede *ser host, net o port*. Indicando respectivamente: máquina (por ejemplo `host 192.168.1.1`) , una red completa (por ejemplo `net 192.168`) o un puerto concreto (por ejemplo `port 22`). Por defecto se asume el tipo host.

dir Especifica desde o hacia donde se va a mirar el flujo de datos. Tenemos src o dst y podemos combinarlos con *or* y *and*. Para el caso de de protocolos punto a punto podemos sustituir por inbound o outbound.Por ejemplo si queremos la dirección de destino 10.10.10.2 y la de origen 192.168.1.2, el filtro será `dst 10.10.10.2 and src 192.168.1.2` . Si se quiere que sea la dirección destino 192.168.1.1 o la dirección origen 192.168.1.2 el código será `dst 192.168.1.1 or src 192.168.1.2`. Pueden seguirse combinando con la ayuda de paréntesis o las palabras *or* y *and*. Si no existe se supone `src or dst`. Por supuesto, esto se puede combinar con los modificadores de tipo anteriores.

proto En este caso es el protocolo que queremos capturar. puede ser `tcp, udp, ip, ether` (en este último caso captura tramas a nivel de enlace), arp (peticiones arp), rarp (peticiones reverse-arp),fddi para redes FDDI.

Estas expresiones siempre pueden combinarse con la ayuda de paréntesis y operadores lógicos, Negación (! not), Concatenación (&& and), Alternativa (| | or).

A continuación se detallan las primitivas que pueden usarse. Lo que aperece entre [ ] es opcional, y el | (pipe) significa `XOR` (.ºexclusiva).

[dst | src] host *¡máquina¿*

Verdadero si la dirección destino u origen del paquete coincide con *¡máquina¿* la cual puede ser una dirección IPv4 (o IPv6 si se ha compilado soporte para el mismo), o un nombre resoluble por el DNS.

Ejemplos:

* Capturar el tráfico originado por la IP 192.168.1.1

```
src host 192.168.1.1
```

* Capturar todo el tráfico cuya dirección origen o destino sea 192.168.1.2

```
host 192.168.1.2
```

ether src | dst | host *¡edir¿*

Este filtro es cierto si la dirección origen (src), la destino (dst) o el cualquiera de las dos(host) coincide con edir. Es obligatorio especificar uno de los tres modificadores.

Ejemplos:

* Capturar el tráfico con destinado a 0:2:a5:ee:ec:10

```
ether dst 0:2:a5:ee:ec:10
```

* Capturar el tráfico cuyo origen o destino sea 0:2:a5:ee:ec:10

```
ether host 0:2:a5:ee:ec:10
```

gateway *¡máquina¿*

Veradadero en caso de que el paquete use *¡máquina¿* como *gateway*. *¡máquina¿* debe estar definida en /etc/ethers y /etc/hosts. Los paquetes capturados por este tipo de filtro, son aquellos cuya dirección Ethernet destino es *¡máquina¿*, pero la dirección IP destino es la de otro sistema.

[dst | src] net *¡red¿*

Verdadero en caso de que la red de la dirección destino, origen o ambas sea *¡red¿*. El parámetro red puede ser una dirección numérica (por ejemplo 192.168.1.0) o bien un nombre resoluble a través de /etc/networks. Resaltar que el uso de net, mask, etc no es compatible con direcciones IPv6. Si se quiere hacer referencia a la red destino se usa `dst` como prefijo. Para la red origen se usa `src`

Ejemplos:

* Capturar todo el tráfico cuya red destino sea 192.168.1.0

```
dst net 192.168.1.0
```

* Capturar todo el tráfico cuya red origen sea 192.168.1.0/28

`src net 192.168.1.0 mask 255.255.255.240` *o también* `src net 192.168.1.0/28`

* Capturar todo el tráfico con origen o destino en la 10.0.0.0/24

`net 10.0.0.0/24` *o también* `net 10.0.0.0 mask 255.255.255.0`

[dst | src] port *¡puerto¿*

Este filtro capturará el paquete en caso de que el puerto (udp o tcp) coincida con *¡puerto¿*. El puerto es un valor numérico entre 0-65535 o bien un nombre resoluble a través del /etc/services.

Ejemplos:

* Capturar todo el tráfico con destino al puerto 23

`dst port 23`

* Capturar todo el tráfico con destino u origen puerto 80

`port 80`

less *¡longitud¿*

Verdadero en caso de que el tamaño del paquete sea menor o igual *¡longitud¿*.

greater *¡longitud¿*

Verdadero en caso de que el tamaño del paquete sea mayor o igual que *¡longitud¿*.

ip proto protocolo

En este caso escucha el protocolo que se le indique. El protocolo puede ser icmp, icmp6, igmp (internet group managent protocol), igrp (interior gateway routing protocol), pim (protocol independent multicast), ah (IP Authentication header), esp (encapsulating security payload), udp o tcp.

Por comodidad disponemos de los alias tcp, udp e icmp que equivalen a ip proto tcp or ip6 proto tcp, etc...

Ejemplos:

* Capturar el todo los paquetes icmp

`ip proto icmp`

* Capturar todo el tráfico udp

```
ip proto udp
```

ip6 protochain *¡protocolo¿* *¡protocolo¿* es el número de prorocolo, este número puede encontrarse en /etc/protocols.

ip protochain protocolo

Igual que el caso anterior pero para IPv4.

ether broadcast

Verdadero si la trama capturada va dirigida hacia la dirección de difusión ethernet.

ip broadcast

Verdadero si el paquete va dirigido a la dirección de difusión de IP.

ether multicast

Verdadero si la trama va dirigida a una dirección multicast ethernet.

ip multicast

Verdadero si el paquete va dirigido a una dirección multicast IP.

ip6 multicast

Verdadero si el paquete va dirigido a una dirección multicast IPv6.

ether proto protocolo

Verdadero si el protocolo que contiene la trama es de tipo protocolo Los protocolos son ip, ip6, arp, rarp, atalk, aarp, decnet, sca, lat, mopdl, moprc e iso.

Pueden utilizarse los siguientes alias para hacer más cómoda la sintaxis: ip, ip6, arp, rarp, aarp, decnet e iso. Estos alias son equivalentes a ether proto ip, ether proto ip6, etc.

Ejemplos:

* Capturar todo tráfico arp

```
ether proto arp o también arp
```

* Capturar todo tráfico ip

```
ether proto ip
```

vlan [vlanid]

Verdadero si la trama capturada es un paquete 802.1Q VLAN. Tener en cuenta que esto cambia el resto de la interpretación del paquete capturado, en especial los desplazamientos a partir de los cuales empiezan a decodificar los protocolos, ya que se asume que se están capturando paquetes que viajan en tramas VLAN. Por último si está presente el parámetro vlanid, sólo se mostrarán aquellos paquetes que vayan a la VLAN vlanid.

Combinando los filtros

Se pueden combinar las expresiones anteriores a través de los operadores not, and y or dando lugar a filtros más complejos. Podemos usar también los equivalentes del lenguaje C: !, && o | |.

Ejemplos:

* Capturar todo el tráfico Web (TCP port 80)

```
tcp and port 80
```

* Capturar el todas las peticiones DNS

```
udp and dst port 53
```

* Capturar el tráfico al puerto telnet o ssh

```
tcp and (port 22 or port 23)
```

* Capturar todo el tráfico excepto el web

```
tcp and not port 80
```

Filtros Avanzados

Podemos crear filtros manualmente, consultando los contenidos de cada uno de los octetos de un paquete, la expresión general para crear un filtro avanzado es:

expr relop expr

Donde:
`relop` puede ser cualquiera de los operadores relacionales de C(¿,¡, ¿= ¡=, = y !=).
`expr` es una expresión aritmética compuesta por una serie de números enteros, los operadores binarios de C, (+, -, *, /, & y |), un operador de longitud, len, y una serie de palabras reservadas que nos permiten el acceso a los diferentes paquetes de datos (ether, fddi, tr, ip, arp, rarp, tcp, udp, icmp e ip6).

Para acceder a los datos dentro de un paquete, se usan los modificadores anteriores y una expresión entera.

Opcionalmente se puede especificar el tamaño de los datos a los que se accede.

protocolo [expr : tam]

Así por ejemplo, el primer byte de la trama ethernet será ether[0], la primera palabra será ether[0:2]. El parámetro tam puede ser 1 (por defecto y no hace falta especificarlo), 2 o 4.

A tener en cuenta: En caso de usar tcp[índice] o udp[índice], implícitamente se aplica una regla para averiguar si es un paquete fragmentado, es decir, usando la notación de estos filtros ip[0:2] & 0x1fff = 0. udp[0] o tcp[0] se refieren al primer byte de la cabecera UDP o TCP.

2.6.3. Funciones específicas

`int pcap_compile(pcap_t *p, struct bpf_program *fp,char *str, int optimize, bpf_u_int32 ne`

Esta funcion se emplea para compilar un programa de filtrado en formato `Tcpdump` (`char* str`) en su BPF equivalente (`bpf_u_int32 netmask`)
Es posible que durante la compilación el programa original se modifique para optimizarlo.
`netmask` es la mascara de la red local, que puede obtenerse llamando a `pcap_lookupnet`.
En caso de que la función retorne -1, se habrá producido un error, para una descripción más detallada de lo sucedido podemos emplear la función pcap_geterr()

`int pcap_setfilter(pcap_t *p, struct bpf_program *fp)`

Una vez compilado el filtro sólo falta aplicarlo, para ello basta con pasar como parametro a esta función el resultado de compilar el filtro con pcap_compile.
En caso de error la función devolverá -1 y puede obtenerse una descripción más detallada

del error con pcap_geterr().

2.6.4. Ejemplo1: Aplicando filtros

```
/*************************************************************************
* file:    pcap_filters.c
* date:    25-Abril-2005
* Author: Alejandro Lopez Monge
*
* Compilacion: gcc -lpcap -o pcap_filters pcap_filters.c
*
* Ejemplo de como filtrar el traico con Libpcap, el programa recibe
* como parametro un filtro, lo compila, lo aplica y se mete en un
* bucle infinito capturando todos los paquetes en modo PROMISCUO
*
*************************************************************************/

#include <pcap.h>
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <netinet/if_ether.h>

//Funcion callback, sera invocada cada vez que se reciba un paquete
void my_callback(u_char *useless,const struct pcap_pkthdr* pkthdr,const u_char* packet)
{
    static int count = 1;
    fprintf(stdout,"%d, ",count);
    fflush(stdout);
    count++;
}

int main(int argc,char **argv)
{
    int i;
    char *dev;
    char errbuf[PCAP_ERRBUF_SIZE];
    pcap_t* descr;
    const u_char *packet;
    struct pcap_pkthdr hdr;
    struct ether_header *eptr; // Ethernet
    struct bpf_program fp;   // contenedor con el programa compilado
    bpf_u_int32 maskp;       // mascara de subred
    bpf_u_int32 netp;        // direccion de red


    if(argc != 2)
     {fprintf(stdout,"Modo de Uso %s \"programa de filtrado\"\n",argv[0]);return 0;}

    dev = pcap_lookupdev(errbuf); //Buscamos un dispositivo del que comenzar la captura
    if(dev == NULL)
     {fprintf(stderr,"%s\n",errbuf); exit(1);}
    else
     {printf("Abriendo %s en modo promiscuo\n",dev);}

    pcap_lookupnet(dev,&netp,&maskp,errbuf); //extraemos la direccion de red y la mascara

    descr = pcap_open_live(dev,BUFSIZ,1,-1,errbuf); //comenzamos la captura en modo promiscuo
```

```
    if(descr == NULL)
     {printf("pcap_open_live(): %s\n",errbuf); exit(1); }

    if(pcap_compile(descr,&fp,argv[1],0,netp) == -1) //compilamos el programa
     {fprintf(stderr,"Error compilando el filtro\n"); exit(1);}

    if(pcap_setfilter(descr,&fp) == -1)              //aplicamos el filtro
     {fprintf(stderr,"Error aplicando el filtro\n"); exit(1);}

    pcap_loop(descr,-1,my_callback,NULL); //entramos en el bucle (infinito)

    return 0;
}
```

2.7. (S3) Interpretando los datos

2.7.1. Organización de un Paquete

Nota: Esta sección sólo hace referencia a el protocolo `Ethernet`.

Cómo recordabamos en la sección introductoria pag 4 cuando una aplicación quiere enviar datos a través de una red, antes tiene que añadir las cabeceras de los protocolo que vaya a emplear en la transmisión. Ver Fig.1 pag 4

En la sección anterior hemos llegado a conseguir un `(u_char *)` con el conjunto de los datos en bruto, también llamados datos `RAW`, pero para poder obtener información inteligíble, tenemos que hacer la labor que la pila de protocolos hubiera hecho por nosotros, es decir extraer e interpretar las cabeceras añadidas por el remitente.

A pesar de que en esta sección se comentan en detalle las estructuras de datos que contendrán las cabeceras, nunca esta de más tener a mano los RFCs más relevantes:

RFC 793 (TCPv4)

RFC 791 (IP)

RFC 768 (UDP)

RFC 826 (ARP)

RFC 792 (ICMPv4)

La primera cabecera que vamos a extraer es la Ethernet, definida en el fichero **ethernet.h** en /usr/includes/net/, nos encontramos con los siguiente:

```
/* This is a name for the 48 bit ethernet address available on many
   systems. */
struct ether_addr
{
  u_int8_t ether_addr_octet[ETH_ALEN];
} __attribute__ ((__packed__));

/* 10Mb/s ethernet header */
struct ether_header
{                                                                    10
  u_int8_t ether_dhost[ETH_ALEN]; /* destination eth addr */
  u_int8_t ether_shost[ETH_ALEN]; /* source ether addr */
  u_int16_t ether_type;                 /* packet type ID field */
} __attribute__ ((__packed__));

/* Ethernet protocol ID's */
#define ETHERTYPE_PUP 0x0200          /* Xerox PUP */
#define ETHERTYPE_IP    0x0800        /* IP */
```

```
#define ETHERTYPE_ARP 0x0806        /* Address resolution */
#define ETHERTYPE_REVARP 0x8035     /* Reverse ARP */

#define ETHER_ADDR_LEN ETH_ALEN  /* size of ethernet addr */
#define ETHER_TYPE_LEN 2         /* bytes in type field */
#define ETHER_CRC_LEN 4          /* bytes in CRC field */
#define ETHER_HDR_LEN ETH_HLEN   /* total octets in header */
#define ETHER_MIN_LEN (ETH_ZLEN + ETHER_CRC_LEN) /* min packet length */
#define ETHER_MAX_LEN (ETH_FRAME_LEN + ETHER_CRC_LEN) /* max packet length */
```

Del código podemos interpretar que el tipo encargado de contener una cabecera Ethernel se llama `ether_header` y que a grandes rasgos tiene 3 datos interesantes: La dirección ethernet origen (`ether_shost`), la dirección ethernet destino (`ether_dhost`) y el tipo de paquete que porta, que puede ser:

ETHERTYPE_PUP Xerox PUP

ETHERTYPE_IP Es un paquete de tipo IP

ETHERTYPE_ARP Es un paquete de tipo ARP (traduccion de direcciones ethernet a ip)

ETHERTYPE_RARP Es un paquete de tipo RARP (traduccion de direcciones ip a ethernet)

En el fichero **netinet/ether.h** están definidas varias funciones útiles para tratar con direcciones Ethernet.

```
//Funciones para convertir una direccion Ethernet de 48bits a texto legibl

extern char *ether_ntoa (__const struct ether_addr *__addr) __THROW;
extern char *ether_ntoa_r (__const struct ether_addr *__addr, char *__buf)
    __THROW;

//Funciones para convertir de formato texto a una direccion Ethernet de 48bits

extern struct ether_addr *ether_aton (__const char *__asc) __THROW;
extern struct ether_addr *ether_aton_r (__const char *__asc,
                                        struct ether_addr *__addr) __THROW;

// Mapeo de un hostname a una direccion Ethernet de 48 bits

extern int ether_hostton (__const char *__hostname, struct ether_addr *__addr)
```

Una vez sabido cómo está estructurada la cabecera Ethernet, podemos extraerla del `u_char * packet`, puesto que sabemos que se corresponde con los 14 primeros bytes (sizeof(struct ether_header)), de hecho los primeros 14 bytes siempre corresponderán a una cabecera Ethernet, no así los siguientes que dependerán del valor que tome el campo `ether_type`.

En este caso vamos a suponer que el paquete es de tipo IP (ether_type=ETHERTYPE_IP), de modo que despueés de los 14 bytes de la cabecera Ethernet, vamos a encontrar una cabecera IP. La estructura que contendrá la cabecera IP está definida en ip.h (/usr/includes/netinet/),

de donde he sacado el siguiente fragmento:

```
/*
 * Structure of an internet header, naked of options.
 */
struct ip
  {
#if __BYTE_ORDER == __LITTLE_ENDIAN
    unsigned int ip_hl:4;           /* header length */
    unsigned int ip_v:4;            /* version */
#endif
#if __BYTE_ORDER == __BIG_ENDIAN
    unsigned int ip_v:4;            /* version */
    unsigned int ip_hl:4;           /* header length */
#endif
    u_int8_t ip_tos;                /* type of service */
    u_short ip_len;                 /* total length */
    u_short ip_id;                  /* identification */
    u_short ip_off;                 /* fragment offset field */
#define IP_RF 0x8000                /* reserved fragment flag */
#define IP_DF 0x4000                /* dont fragment flag */
#define IP_MF 0x2000                /* more fragments flag */
#define IP_OFFMASK 0x1fff           /* mask for fragmenting bits */
    u_int8_t ip_ttl;                /* time to live */
    u_int8_t ip_p;                  /* protocol */
    u_short ip_sum;                 /* checksum */
    struct in_addr ip_src, ip_dst;  /* source and dest address */
  };
/*
 * Definitions for options.
 */
#define IPOPT_COPY      0x80
#define IPOPT_CLASS_MASK 0x60
#define IPOPT_NUMBER_MASK 0x1f

#define IPOPT_COPIED(o) ((o) & IPOPT_COPY)
#define IPOPT_CLASS(o)  ((o) & IPOPT_CLASS_MASK)
#define IPOPT_NUMBER(o) ((o) & IPOPT_NUMBER_MASK)

#define IPOPT_CONTROL 0x00
#define IPOPT_RESERVED1 0x20
#define IPOPT_DEBMEAS 0x40
#define IPOPT_MEASUREMENT IPOPT_DEBMEAS
#define IPOPT_RESERVED2 0x60

#define IPOPT_EOL       0           /* end of option list */
#define IPOPT_END       IPOPT_EOL
#define IPOPT_NOP       1           /* no operation */
#define IPOPT_NOOP      IPOPT_NOP

#define IPOPT_RR        7           /* record packet route */
#define IPOPT_TS        68          /* timestamp */
#define IPOPT_TIMESTAMP IPOPT_TS
#define IPOPT_SECURITY 130          /* provide s,c,h,tcc */
#define IPOPT_SEC       IPOPT_SECURITY
#define IPOPT_LSRR      131         /* loose source route */
#define IPOPT_SATID     136         /* satnet id */
#define IPOPT_SID       IPOPT_SATID
```

```
#define IPOPT_SSRR      137             /* strict source route */
#define IPOPT_RA        148             /* router alert */

//BSD Flavour                                                          60

struct iphdr
  {
#if __BYTE_ORDER == __LITTLE_ENDIAN
    unsigned int ihl:4;
    unsigned int version:4;
#elif __BYTE_ORDER == __BIG_ENDIAN
    unsigned int version:4;
    unsigned int ihl:4;
#else                                                                  70
# error "Please fix <bits/endian.h>"
#endif
    u_int8_t tos;
    u_int16_t tot_len;
    u_int16_t id;
    u_int16_t frag_off;
    u_int8_t ttl;
    u_int8_t protocol;
    u_int16_t check;
    u_int32_t saddr;                                                   80
    u_int32_t daddr;
    /*The options start here. */
  };
```

Cómo podeis ver no hay una única definición de contenedor para una cabecera IP, como se dice en el argot, *hay varios flavours (sabores)*, tenemos BSD flavorus, Linux Flavours... Son distintas de maneras de organizar los mismos datos, así que para gustos ... ahora también las cabeceras IP.

No voy a entrar a describir en detalle los contenidos de la cabecera IP, para eso están los RFC's , sin embargo quiero hacer especial hincapié en el campo **protocol** ya que nos servirá para determinar cual sera la tercera y última cabecera a extraer. Existen multitud de protocolos, aunque los más importantes son ICMP (protocolo=1) TCP (protocol=6) UDP (protocol=17) IPV6 (protocol=41).

En el siguiente ejemplo se muestra cómo llegar a extraer el payload de un paquete TCP (IPheader.protocol=6), si no pongo un ejemplo por cada tipo de protocolo es por que sería bastante repetitivo. Sea cual sea el protocolo los pasos a seguir son estos:

1. Extraemos la cabecera Ethernet (6 primeros bytes).
2. Consultamos el campo ether_type, para saber si es IP.
3. Si es IP, consultamos el campo protocol.
4. Vamos a /usr/includes/netinet y consultamos cómo es la cabecera para ese protocolo.

5. Consultamos el tamaño de esta última cabecera y el payload, si es que lo lleva, va a continuación.

2.7.2. Ejemplo1. Extracción del payload de un paquete TCP

```
/*************************************************************************
* file:    pcap_looktcp.c
* date:   25-Abril-2005
* Author: Alejandro Lopez Monge
*
* Compilacion: gcc -lpcap -o pcap_looktcp pcap_looktcp.c
*
* Ejemplo de como extraer el payload de una paeticion WEB port=80
*
*************************************************************************/

#include <pcap.h>
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <netinet/if_ether.h>
#include <netinet/ether.h>
#include <netinet/ip.h>

//Funcion callback, sera invocada cada vez que se reciba un paquete
void my_callback(u_char *useless,const struct pcap_pkthdr* pkthdr,const u_char* packet)
{
    static int count = 0; //inicializamos el contador
    count ++;printf("\n");
    struct ether_header *eptr;

   /* Apuntamos el puntero a la cabecera Ethernet al
      comienzo del paquete
    */
    eptr = (struct ether_header *) packet;
    printf("Paquete numero: %d\n",count);
    printf("MAC origen: %s\n", ether_ntoa(eptr->ether_shost) );
    printf("MAC destino: %s\n", ether_ntoa(eptr->ether_dhost) );

    //Comprobamos de que tipo es el paquete

    if (ntohs (eptr->ether_type) == ETHERTYPE_IP)
    {printf("Es de tipo IP, por ahora nos vale\n");}
    else if (ntohs (eptr->ether_type) == ETHERTYPE_ARP)
    {printf("Es de tipo ARP, no nos vale\n"); return;}
    else if (ntohs (eptr->ether_type) == ETHERTYPE_REVARP)
    {printf("Es de tipo RARP, no nos vale\n"); return;}
    else
    {printf("Es de tipo desconocido, no nos vale\n");}

    /* Ahora extraemos la cabecera IP, por lo que tenemos
       que desplazar el tamaño de la cabecera Ethernet ya
       procesada
    */
    struct ip *ipc;
    ipc=packet+sizeof(struct ether_header);
    printf("El ttl es %d\n",ipc->ip_ttl);
    printf("IP origen: %s\n",inet_ntoa(ipc->ip_src));
    printf("IP destino: %s\n",inet_ntoa(ipc->ip_dst));
```

```
    /* Comprobamos que el protocolo sea TCP */
    switch (ipc->ip_p)
    {
     case 1:
      {printf("Es ICMP, no vale\n");return;}
     case 6:
      {printf("Es TCP, nos vale\n");break;}
     case 17:
      {printf("Es UDP, no nos vale\n");return;}
     default:
      {printf("Es un protocolo desconocido, no nos vale\n");return;}
     }
     char* payload= packet+sizeof(struct ether_header)+ipc->ip_len;
     printf("Payload: \n%s\n",payload);
}

int main(int argc,char **argv)
{
    char *filtro="tcp and port 80"; //solo el trafico web
    char *dev;
    char errbuf[PCAP_ERRBUF_SIZE];
    pcap_t* descr;
    const u_char *packet;
    struct pcap_pkthdr hdr;
    struct ether_header *eptr; // Ethernet
    bpf_u_int32 maskp;         // mascara de subred
    bpf_u_int32 netp;          // direccion de red
    struct bpf_program fp;   // El programa de filtrado compilado

    dev = pcap_lookupdev(errbuf); //Buscamos un dispositivo del que comenzar la captura
    if(dev == NULL)
     {fprintf(stderr,"%s\n",errbuf); exit(1);}
    else
     {printf("Abriendo %s en modo promiscuo\n",dev);}

    pcap_lookupnet(dev,&netp,&maskp,errbuf); //extraemos la direccion de red y la mascara

    descr = pcap_open_live(dev,BUFSIZ,1,-1,errbuf); //comenzamos la captura en modo promiscuo

    if(pcap_compile(descr,&fp,filtro,0,netp) == -1) //compilamos el programa
     {fprintf(stderr,"Error compilando el filtro\n"); exit(1);}

    if(pcap_setfilter(descr,&fp) == -1)            //aplicamos el filtro
     {fprintf(stderr,"Error aplicando el filtro\n"); exit(1);}

    if(descr == NULL)
     {printf("pcap_open_live(): %s\n",errbuf); exit(1); }

    pcap_loop(descr,-1,my_callback,NULL); //entramos en el bucle (infinito)

    return 0;
}
```

2.8. (S4) Volcando los datos a un fichero

`Libpcap` nos ofrece también la posibilidad de guardar los datos en un fichero para procesarlos más adelante, (lo que en ingles se llama *dump to a file*) vamos a ver cuales son las funciones específicas para esta tarea.

2.8.1. Funciones específicas

`pcap_dumper_t *pcap_dump_open(pcap_t *p, char *fname)`
Esta función se utiliza para abrir un fichero de salida en el que guardar los datos que vayamos capturando. Si todo va bien la función abrirá el fichero `char* fname` en modo escritura y nos devolverá un puntero a un descriptor de tipo `pcap_dumper_t` sobre el que podremos comenzar a volcar datos. En caso de error la función devolverá `NULL` y podemos ver un error más descriptivo con la función `pcap_geterr()`.

`pcap_open_offline(filename, errbuf))`
Esta función sirve para abrir un fichero con paquetes ya guardados en formato `TCPDUMP` en modo lectura. Los parámetros son muy parecidos a los de la función anterior, `fname` para la ruta del fichero, `NULL` en caso de que la función falle, la diferencia es que en este caso para consultar la descripción del error consultaremos `errbuf`.

`void pcap_dump(u_char *user, struct pcap_pkthdr *h,u_char *sp)`
Una abierto el fichero salida con pcap_dump_open(), podemos comenzar a loguear los paquetes con esta función.

`void pcap_dump_close(pcap_dumper_t *p)`
Si hemos terminado con el fichero de salida, podemos cerrarlo llamando a esta función.

2.8.2. Ejemplo1: Guardando los datos

```
/************************************************************************
* file:     dump_packets.c
* date:   25-Abril-2005
* Author: Alejandro Lopez Monge
*
* Compilacion: gcc -lpcap -o dump_packets dump_packets.c
*
* Este ejemplo muestra como capturar paquetes y guardarlos en un fichero
* para su posterior procesamiento, todo ello empleando las funciones propias
* de la API de Libpcap. Combinamos varias de las cosas que ya hemos aprendido
* aplicar filtros, este programa solo captura los paquetes que sean de la misma
* red y que vayan destinados al puerto 23, y una cosa nueva las estadisticas.
************************************************************************/

#include <unistd.h>
#include <stdio.h>
#include <pcap.h>
#include <netinet/in.h>
#include <sys/socket.h>

#define IFSZ 16
#define FLTRSZ 120
#define MAXHOSTSZ 256
#define PCAP_SAVEFILE "./pcap_savefile"

extern char *inet_ntoa();

int usage(char *progname)
{
        printf("Uso: %s <interfaz> [<fichero salida>]\n", basename(progname));
        exit(11);
}

int main(int argc, char **argv)
{
        pcap_t *p;
        struct pcap_stat ps;  /* estadisticas */
        pcap_dumper_t *pd;  /* dump file */
        char ifname[IFSZ];     /* nombre de la interfaz */
        char filename[80];     /* nombre del dump file*/
        char errbuf[PCAP_ERRBUF_SIZE]; /* descripcion del error */
        char lhost[MAXHOSTSZ]; /* nombre del localhost */
        char fltstr[FLTRSZ];  /* texto del bpf filter*/
        char prestr[80];       /* prefijo para los errores */
        struct bpf_program prog; /* BPF compilado */
        int optimize = 1;
        int snaplen = 80;      /* Tamaño por paquete */
        int promisc = 0;       /* modo de captura */

        int to_ms = 1000;      /* timeout*/
        int count = 20;        /* numero de paquetes a capturar */
        int net = 0;        /* direccion de red*/
        int mask = 0;      /* mascara de subred */
        char netstr[INET_ADDRSTRLEN]; /* direccion de red en modo texto*/
        char maskstr[INET_ADDRSTRLEN]; /* mascara de red en modo texto */
        int linktype = 0;       /* tipo de enlace de datos*/
        int pcount = 0;        /* Numero de paquetes leidos */
```

```
if (argc < 2)
        usage(argv[0]);                                                    60

if (strlen(argv[1]) > IFSZ) {
        fprintf(stderr, "Nombre de interfaz invalido.\n");exit(1);
}
strcpy(ifname, argv[1]);

/*
 * Si se especifica un nombre para el fichero salida, se usara. En caso
 * contrario se usara uno por defecto
 */                                                                        70
if (argc >= 3)
        strcpy(filename,argv[2]);
else
        strcpy(filename, PCAP_SAVEFILE);

if (!(p = pcap_open_live(ifname, snaplen, promisc, to_ms, errbuf))) {
        fprintf(stderr, "Error al abrir la interfaz%s: %s\n",ifname, errbuf);
        exit(2);
}
                                                                           80
if (pcap_lookupnet(ifname, &net, &mask, errbuf) < 0) {
        fprintf(stderr, "Error looking up network: %s\n", errbuf);
        exit(3);
}

/* Obtenemos el nombre del localhost para aplicarlo en el filtro */
if (gethostname(lhost,sizeof(lhost)) < 0) {
        fprintf(stderr, "Error consultado el hostname.\n");
        exit(4);
}                                                                          90

inet_ntop(AF_INET, (char*) &net, netstr, sizeof netstr);
inet_ntop(AF_INET, (char*) &mask, maskstr, sizeof maskstr);

/* Aplicamos el siguiente filtro */
sprintf(fltstr,"dst host %s and src net %s mask %s and tcp port 23",lhost, netstr, maskstr);

/* Lo compilamos */
if (pcap_compile(p,&prog,fltstr,optimize,mask) < 0) {
        fprintf(stderr, "Error compilando el filtro%s: %s\n",ifname, pcap_geterr(p));   100
        exit(5);
}

/* Cargamos el filtro */
if (pcap_setfilter(p, &prog) < 0) {
        pcap_perror(p,prestr);
        exit(6);
}

/* Comenzamos el DUMP */                                                   110
if ((pd = pcap_dump_open(p,filename)) == NULL) {
        fprintf(stderr,"Error abriendo el fichero \"%s\" para escritura: %s\n",filename, pcap_geterr(p));
        exit(7);
}

if ((pcount = pcap_dispatch(p, count, &pcap_dump, (char *)pd)) < 0) {
```

```
        pcap_perror(p,prestr);
        exit(8);
    }
    printf("Numero de paquetes correctamente procesados: %d.\n",pcount);

    if (!(linktype = pcap_datalink(p))) {
        fprintf(stderr,"Error obteniendo el data link %s",ifname);
        exit(9);
    }
    printf("El data link es %s: %d.\n",ifname, linktype);

    //imprimimos las estadisticas
                    if (pcap_stats(p, &ps) != 0) {
        fprintf(stderr, "Error obteniendo las estadisticas: %s\n",pcap_geterr(p));
        exit(10);
    }

    /* Estas son las estadisticas */
    printf("Estadisticas:\n");
    printf(" %d Numero de paquetes que han pasado el filtro\n", ps.ps_recv);
    printf(" %d Numero de paquetes que no han llegado a salir del Kernel\n", ps.ps_drop);

    pcap_dump_close(pd);
    pcap_close(p);
}
```

2.8.3. Ejemplo2: Recuperando los datos

```
/*************************************************************
*
* Fichero: pcap_read.c
* Fecha:  Alejandro Lopez Monge
*
* Compilacion: gcc -lpcap -o pcap_read lsdevs.c
*
* Programa original: http://publib.boulder.ibm.com/
* Descripcion:
*  Abrirmos un fichero donde previamente hemos volcado paquetes
*  con las funciones pcap_dump y los recuperamos como si los estuviesemos
*  leyendo de la red en tiempo real
*
*************************************************************/
#include <stdio.h>
#include <pcap.h>
#define IFSZ 16
#define FLTRSZ 120
#define MAXHOSTSZ 256
#define PCAP_SAVEFILE "./pcap_savefile" //ruta donde se salar"s el fichero por defecto
int packets = 0; //Contador de paquetes

//modo de empleo
int usage(char *progname)
{
        printf("Uso: %s <interfaz> [<fichero entrada>]\n", basename(progname));
        exit(7);
}

// print_addrs() escribe las direccions IP origen y destino del paquete al stdout
void print_addrs(u_char *user, const struct pcap_pkthdr *hdr, const u_char *data)
{
        int offset = 26; // 14 MAC header + 12 IP
        if (hdr->caplen < 30) {
                // Los datos capturados no son suficientes para extraer la dirección IP
                fprintf(stderr,"Error: El paquete capturado no es lo suficientemente grande para extraer direcciones IP.\n"
                return;
        }
        printf("Paquete recivido desde: %d. %d. %d. %d\n",
                data[offset], data[offset+1], data[offset+2], data[offset+3]);
        if (hdr->caplen >= 34) {
                printf("y destinado a %d. %d. %d. %d\n",
                        data[offset+4], data[offset+5],
                        data[offset+6], data[offset+7]);
        }
        packets++;
}

int main(int argc, char **argv)
{
        pcap_t *p;
        char ifname[IFSZ];      //nombre de la interfaz
        char filename[80];      //nombre del savefile a leer
        char errbuf[PCAP_ERRBUF_SIZE];
        char prestr[80];
        int majver = 0, minver = 0; // versiones
```

```
        // El nombre de la interfaz debe ser pasado por línea de comandos.
        if (argc < 2)
            usage(argv[0]);                                                    60

        if (strlen(argv[1]) > IFSZ) {
                fprintf(stderr, "Nombre de interfaz invalido\n");
                exit(1);
        }
        strcpy(ifname, argv[1]);

         /* Si no hay un segundo argumento (nombre fichero entrada)
          *  se usa el nombre por defecto "./pcap_savefile"
          */                                                                   70
        if (argc >= 3)
                strcpy(filename,argv[2]);
        else
                strcpy(filename, PCAP_SAVEFILE);
         /* Abre la intefaz para la captura de datos.
          * Debe llamarse antes de capturar cualquier paquete
          */
        if (!(p = pcap_open_offline(filename, errbuf))) {
                fprintf(stderr,"Error abriendo el fichero, %s, en modo lectura: %s\n",
                        filename, errbuf);                                      80
                exit(2);
        }

         //pcap_dispach(), por cada paquete lee del savefile, hasta error o EOF
        if (pcap_dispatch(p, 0, &print_addrs, (char *)0) < 0) {

                sprintf(prestr,"Error, no puedo leer de la interfaz %s",ifname);
                pcap_perror(p,prestr);
                exit(4);
        }                                                                       90

        printf("\nPaquetes leidos: %d\n", packets);

                                // Mostramos las versiones en pantalla
        if (!(majver = pcap_major_version(p))) {
                fprintf(stderr,"Error obteniendo la -mayor version- de la interfaz: %s",ifname);
                exit(5);
        }
        printf("La -mayor version- usada para crear el fichero es: %d.\n", majver);
        if (!(minver = pcap_minor_version(p))) {                                100
                fprintf(stderr,"Error obteniendo la -minor version- de la interfaz: %s",ifname);
                exit(6);
        }
        printf("La -minor version-: %d.\n", minver);
        pcap_close(p); // Cerramos el dispositivo de captura y la memoria usada por el descriptor
}
```

2.9. Estructuras de datos usadas por Pcap

2.9.1. ¿Dónde están definidas?

Dependiendo del sistema operativo en el que estés programando, el fichero `pcap.h` estará ubicado en un lugar u otro (en Linux por defecto está en `/usr/include/`. Este fichero contiene las cabeceras de todas las funciones y las definciones de las estructuras de datos específicas de `Libpcap`, a continuación se detallan las más relevantes.

2.9.2. Principales Estructuras

Informacion de Interfaces

```
/*
  La llamada a pcap_findalldevs, construye una lista enlazada de
  pcap_if (pcap_if=pcap_if_t), en la cual se recpge toda la infor
  macion de cada una de las interfaces de red instaladas.
*/

struct pcap_if {
        struct pcap_if *next; // enlace a la siguiente definicion de interfaz
        char *name;           // nombre de la interfaz (eth0,wlan0...)
        char *description;    // descripcion de la interfaz o NULL
        struct pcap_addr *addresses;// lista enlazada de direcciones asociadas a esta interfaz
        u_int flags;          // PCAP_IF_ interface flags
};

/*
   Una interfaz puede tener varias direcciones, para contenerlas todas
   se crear una lista con un pcap_addr por cada una
*/

struct pcap_addr {
        struct pcap_addr *next; // enlace a la siguiente direccion
        struct sockaddr *addr;   // direccion
        struct sockaddr *netmask; // mascara de red
        struct sockaddr *broadaddr; // direccion de broadcast para esa direccion
        struct sockaddr *dstaddr; // direccion de destino P2P (punto a punto)
};
```

Estadisticas

```
/*
  Mediantes la llamada a pcap_stats obtenemos la siguiente
  estructura con valiosa informacion estadistica
*/

struct pcap_stat {
        u_int ps_recv;          // numero de paquetes recibidos
        u_int ps_drop;          // numero de paquetes dropped
        u_int ps_ifdrop;        // drops por cada interfaz (aun no soportado)
};
```

Paquetes

```
/*
  Cada paquete del dump file va precedido por esta cabecera generica
  de modo que un mismo fichero puede contener paquetes heterogeneos
*/

struct pcap_pkthdr {                                                    50
        struct timeval ts;      // time stamp (marca de tiempo)
        bpf_u_int32 caplen;     // tamano del paquete al ser capturado
        bpf_u_int32 len;        // tamano real del paquete en el fichero;
};

struct pcap_file_header {
        bpf_u_int32 magic;
        u_short version_major;
        u_short version_minor;
        bpf_int32 thiszone;     // correcion de GMT a local              60
        bpf_u_int32 sigfigs;    // precision de los timestamps
        bpf_u_int32 snaplen;    // tamano maximo salvado de cada paquete
        bpf_u_int32 linktype;   // tipo de DataLink
};
```

La estructura pcap_t

```
/*
  La estructura pcap_t que aparece en muchas funciones, es una estructura
  opaca para el usuario. Por lo tanto no es necesario que sepa cuales   70
  son sus contenidos, basta con que se sepa que ahi es donde Libpcap
  guarda sus variables internas
*/
```

Referencias

[**http://publib.boulder.ibm.com**] Mucho del código fuente empleado en este tutorial, lo he sacado de esta página, especialmente los ejemplos en los que se explica cómo volcar la información a un fichero y cómo recuperarla después.

[**http://www.cet.nau.edu/ mc8/Socket/Tutorials/section1.html**] El primero de los dos tutoriales mencionados en la página web de TCPDUMP.

[**http://reactor-core.org/libpcap-tutorial.html**] Este es el segundo.

[**www.grid.unina.it/software/ ITG/D-ITGpubblications/TR-DIS-122004.pdf**] Sniffer programado con Libpcap.

[**www.tcpdump.org/papers/bpf-usenix93.pdf**] Un gran documento del que aprendí que es cómo funcionan los Packet Filters, recomiendo su lectura.