v.1.3.7 - 6/3/18

Practical Exercises in Computer Networks

Programmatic access to the datalink layer

© 2015-18, José María Foces Morán

Computer networks are complex systems. Abstractions play a central role in the conceptualization required to design and manage whereat the *network architecture* concept stands out conspicuously.

Network architectures expose concepts by way of interfaces, thus enabling an efficient use of our intellectual prowess, focusing attention on relevant details. Interfaces are abstractions themselves; their different forms provide controlled access to resources that, by their own critical nature, entail precisely controlled access in order to guarantee their smooth functioning. Networks are made of computer systems -note *systems*, not simply computers but computer systems, with all their seasoning of operating system, systems software and application software.

Network access in operating systems of today is essential, consequently, the systems software responsible for providing that access is what constitutes the *network software stack or, more simply, stack.* The network stack is implemented deep into the core of operating systems so as to guarantee as high a level of performance as attainable. Networking: on the one hand we have the architecture and its interfaces, on the other hand we have the fact that its implementation lives *within* the operating system. For instance, how can we program at the layer 2 service interface? What is our role when doing such programming? *Am I to use the hat* of a systems programmer or that of an application programmer? This practical provides aims to set these in perspective by programming a simple, layer-2 access user *application*.

The Linux network stack and the datalink

The Linux kernel, among a million other responsibilities, implements the whole network stack alongside their service interfaces (The different types of Linux sockets). All the networking data structures, algorithms and the external access APIs are included in the kernel and they all run in supervisor mode[1]. If you were in charge of writing a new network protocol, you would have to build an *extension* of the Linux kernel, probably in the form of a *Linux protocol handler*. For the purpose of quickly introducing you to programming against the datalink service interface, we will program a user-space application since it results substantially lighter in terms of technical detail than the process of composing a full fledged Linux protocol handler.

The specific datalink we are going to consider is the ubiquitous and famous Ethernet which is implemented partly on the Network Interface Card's hardware and software and on the NIC driver. The driver receives requests from the kernel and translates those requests into a series of sophisticated time-dependent actions

[1] **Supervisor mode** is the name of the microprocessor mode of operation that provides tasks its highest level of privilege and is the mode in which the Linux kernel is executed. By contrast, applications are run in **user mode.** For more details, you may want to consult an Operating Systems textbook.

that confer drivers their mystifying nature; the NIC driver registers an ISR (Interrupt Service Routine), thereon, when a new datalink frame is received in the NIC's receive frame buffer, the NIC asserts its specific interrupt line, thereby causing the Interrupt Controller to interrupt the processor and, eventually triggering the receive routine of the ISR. All this action takes place within the kernel realm, however, for the time being, we may only afford building a user-space application.

Access to the datalink Service Interface (SIF)

We have several choices to access the native datalink in Linux and UNIX systems, among them we may cite:

- UNIX DLPI (Data Link Provider Interface)
- BPF (Berkeley Packet Filter)
- The old SOCK_PACKET interface of Linux
- Today's AF_PACKET interface of Linux

Libpcap is a portable library for accessing the *datalink packet capturing functionality* of the host operating system. Note that DLPI, BPF, SOCK_PACKET and AF_PACKET are os-native datalink service interfaces, by contrast, libpcap is a user-space, library-based interface to each one of those service interfaces. libpcap is meant to offer portability across the systems mentioned above, including Windows. Let's read from the Linux man page for **packet**:

> **$ man packet:** Packet sockets are used to receive or send raw packets at the device driver **(OSI Layer 2)** level. They allow the user to implement **protocol modules in user space** on top of the physical layer.

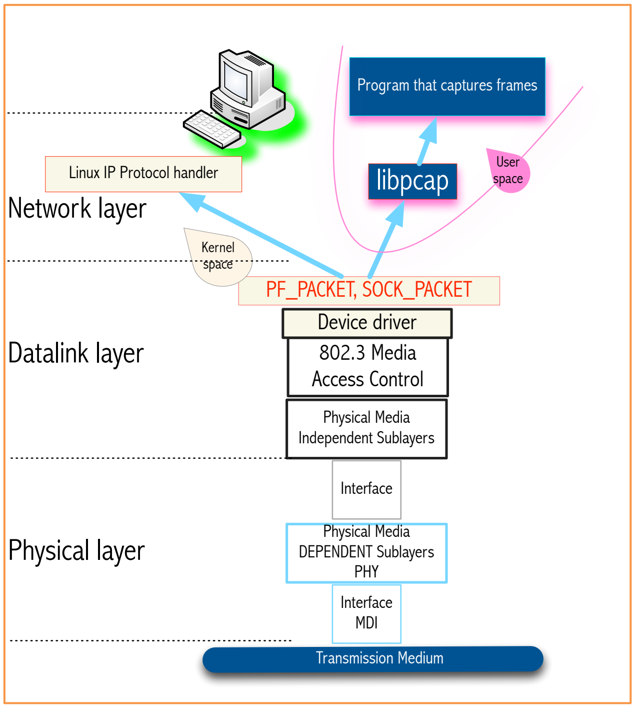
Program that captures frames
libpcap
User space
Linux IP Protocol handler
Network layer
Kernel space
PF_PACKET, SOCK_PACKET
Device driver
802.3 Media Access Control
Datalink layer
Physical Media Independent Sublayers
Interface
Physical Media DEPENDENT Sublayers PHY
Physical layer
Interface MDI
Transmission Medium

Fig. 1: Partial protocol graph illustrating libpcap's spot in a Linux network stack

libpcap installation in Linux

Libpcap is implemented as a dynamic library which installation is straightforward in Linux and other systems such as Windows (Winpcap) and OS-X. You may follow the following procedure to have libpcap installed in your Linux (Debian or Ubuntu). In general, before installing the libpcap-dev package, you should resynchronize the Ubuntu package overviews (Option update of apt-get; see below); then, you can update the packages installed in your system per the procedure explained right below. Take into account that the update operation might take a substantial amount of time, so you might install the new package directly, which, in a properly managed Linux, should cause no problems:

```
$ sudo apt-get update
$ sudo apt-get upgrade
$ sudo apt-get install libpcap-dev
```

Once the library has been successfully installed, download the following libpcap test program and compile it[2]:

```
$ wget http://paloalto.unileon.es/ngn/labs/NGN-Lab-Libpcap--simpleCapturePackets.c

$ gcc -o simpleCapture NGN-Lab-Libpcap--simpleCapturePackets.c –lpcap
```

The program just built accesses the layer-2 indirectly, by way of libpcap, nevertheless it may receive all the traffic passing through your machine's layer 2, and that might be considered harmful for a variety of reasons including the ability to scan through other system's users data. Consequently, in order to execute the layer-2 accessing program you will need to elevate your privileges by issuing the command with sudo as in:

```
$ sudo ./simpleCapture 5
```

This program will capture 5 packets and printout a number for each one; if you miss sudo, the program will issue a number of errors accessing the relevant system network device file. Maybe, at the time you are executing the resulting executable (capturePackets) no traffic is crossing your machine’s layer 2, consequently you’ll have to generate some, you may do so by sending a few pings from another nearby host to yours; quickly the frame numbers will appear on the terminal you ran capturePackets on. The type of networking equipment used in Lab B6 consists basically of Ethernet switches, thus, the only traffic your host will receive consists of unicast traffic directed to it and broadcast traffic (Frames which destination MAC address is the all-ones address or 0xffffffffffff); switches flood frames sent to the broadcast address, thereby guaranteeing that all the hosts belonging to the LAN receive it. The program will consequently inform us about the arp requests being resolved now, which are carried into Ethernet broadcast frames and which are the result of the hosts attempts of sending packets to their default router, for internet access or, sending those packets to hosts belonging in the local network. (A file server, for example).

[2] **Compilation on** various versions of **MAC OS-X** systems is exactly like the procedure just explained, however, some of the pcap calls may be require specific parameter values; for example the call pcap_open_live(netDevice, BUFSIZ, 0, timeout, errbuf) requires the timeout to be greater than 0, by contrast to Linux which accepts 0 or -1. Differences among systems happen quite often.

The quintessential libpcap program

To get started with libpcap, probably, the best is to write a short test program that convince us that working with it is straightforward. Donwload the following C program which uses libpcap to capture a number of frames and printout some of their contents. After compiling the program and running it, you'll be much more confident using the library. By practicing, we aim to emulate the celebrated QEC note taking method[3].

1. Download the proposed, simple test program:

```
$ wget http://paloalto.unileon.es/ngn/labs/NGN-Lab-Libpcap--test2.c
```

2. Download

```
 $ gcc -o test2 NGN-Lab-Libpcap--test2.c -lpcap
```

(A few warnings will be issued by the compiler, however, pay no attention to them for the time being. If you receive errors, probably they have been caused by a defective installation of libpcap; you will have to attempt the installation again and check every step is all right).

The program assumes that your Linux is connected to the lab's LAN via the wired connection (Interface Ethernet 0, eth0, for example), otherwise, your Linux may select another device like Bluetooth on which no traffic will be seen, then, please make sure you connect your Ethernet to the lab's net. If you will, you may fix the name of the network device on which you wish the program to capture frames from, to that end, in function performCapture(), replace the following line for the line appearing next to it (In blue):

~~defaultNetDevice = pcap_lookupdev(errbuf);~~

```
defaultNetDevice = "eth0";
```

Now, please, make sure you connect your Linux's eth0 NIC to the lab's network. Modern versions of Linux follow Ethernet device naming conventions other than the classical eth[n] that we just mentioned, therefore, the safest procedure to make sure that your program uses the correct device consists of browsing the listing resulting from `$ ifconfig` and selecting the right device.

The program captures the number of frames entered by the user on the first command line argument, then it will capture the frames and print out the MAC addresses included in each of them (You may get a glimpse at the fields of an Ethernet frame at the beginning of the Exercises section on page 6):

```
jose@josephus:~$ sudo ./test2 5
[sudo] password for jose:

Network device name = eth0
IP address = 192.168.2.0
Network mask = 255.255.255.0
```

[3] **QEC stands for Question-Evidence-Conclusion:** A method of note taking in nontechnical courses proposed by Calvin Newport in his book "How To Become a Straight-A Student". In passing, I should comment that in technical courses the method of preference consists of solving *mega problem sets,* i.e. practice, practice, practice!

```
Capturing 5 frames:
Packet captured no. 0 Header length: 42
Destination MAC: 70:56:81:c4:df:ba:
Source MAC: 10:fe:ed:02:48:06:
Ethertype:
CRC:

Packet captured no. 1 Header length: 60
Destination MAC: 10:fe:ed:02:48:06:
Source MAC: 70:56:81:c4:df:ba:
Ethertype:
CRC:

Packet captured no. 2 Header length: 343
Destination MAC: 10:fe:ed:02:48:06:
Source MAC: 70:56:81:c4:df:ba:
Ethertype:
CRC:

Packet captured no. 3 Header length: 428
Destination MAC: 70:56:81:c4:df:ba:
Source MAC: 10:fe:ed:02:48:06:
Ethertype:
CRC:

Packet captured no. 4 Header length: 66
Destination MAC: 10:fe:ed:02:48:06:
Source MAC: 70:56:81:c4:df:ba:
Ethertype:
CRC:

Finished. 5 frames captured.
```

By simple inspection you'll see that the program's output is not complete, in particular, the Ethernet frame's Ethertype and CRC fields are not printed out: that is one of the activities I will suggest you later, after we briefly study the programm's use of libpcap.

An example flow of calls in libpcap (NGN-Lab-Libpcap--test2.c)

Function performCapture() articulates the main flow of the program:

1. It seeks a capturable network device in your system:
   pcap_lookupdev()

2. It prints the network device's IP and Netmask by invoking the program's **printIPandMask()** function which invokes libpcap's:
   pcap_lookupnet()
   inet_ntoa(): Turn an inet address into an ASCII, printable string

3. It opens the network device (Usually, it will be eth0):
   pcap_open_live(): This call returns a pointer to an instance of a pcap_t type that will be used in the ensuing pcap call: **pcap_loop()**

4. The function call to **pcap_loop()** specifies the program's function getNewFrame() as the callback function that will be called by pcap whenever a new datalink frame passes through the interface (It is transmitted or received).

Please, refer to **NGN-Lab-Libpcap--test2.c** source listing at appendix 1 as you follow the above sequential program flow; the source code includes a few comments that may provide you answers to your initial questions, nevertheless, for a more thorough perspective you will want to visit libpcap's web site where you will find lots of tutorials, documentation, the library *bits* themselves:

http://www.tcpdump.org/

Suggested exercises

1. Extend the provided libpcap test (**NGN-Lab-Libpcap--test2.c**) program so that it be able to carry out the following operations:

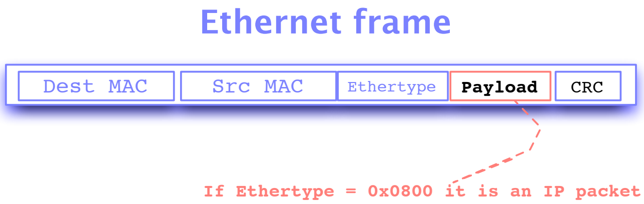

Fig. 2: Ethernet frame

a. Instead of working with the default network device, **have your program capture the specific network device name to use**, from the command line. You can obtain a listing of your currently configured network devices by issuing a `ifconfig` command, without any arguments. If your host has more than one network device active at a time, your rewritten program should be able to select which device's traffic to display on the screen.

b. **Print out the Ethertype of each received frame**. The Ethertype is a TCP/IP multiplexing key as we studied in Computer Networks. This field is a 16-bit unsigned integer which bytes will have to be reordered to a format consistent with your machine's byte ordering: Network Byte Order is Big Endian and, your machine if equipped with an Intel x86 processor will be Little Endian, thereby leading your program to misinterpret the frame's multi-byte fields. In order to make your program portable across all platforms, use one of functions of the ntoh family: ntohs(), ntohl(), etc. The Linux man command may help you with this:

```
$ man ntoh
```

c. **Print out the CRC byte by byte in hex**. In this case, since we are interested in each of the

CRC field bytes, it is not necessary that you call any of ntoh() calls. *Is this possible at all?* Can we access a new frame's CRC field? Think again!

   d. Mark the frames which destination address is the **broadcast address** by printing them out along **with an asterisk**. Search for a few examples of protocols that send broadcast traffic (arp is an important protocol that uses broadcast frames to encapsulate arp requests).

2. Provide a plausible **explanation** about the workings of the **pcap_loop()** call; probably you want to suggest how it is implemented by providing us some sketch C code. The implementation must necessarily include an asynchronous mechanism of communication between the NIC and the Linux kernel. This mechanism, usually involves some hardware interrupt processing.

3. In this exercise I suggest that you **peek at the frame's payload contents**, first, we want to delimit the payload so that it results easier to handle it programmatically, how would you do this? (Hint: we want to know exactly where in the frame the payload begins and where it ends. You will have to use the frame's size and your knowledge of the frame's structure).

   a. In fact, print out the **frame's payload**. For printing the payload, you'll emulate the behavior of the Ethernet procotol by first deencapsulating the payload and then printing it out as though your program were the protocol demultiplexed (Ethernet's multiplexing key is known as Ethertype and is 16-bit wide).

4. Have your program select the **frames that contain an IP packet**, you may use **ping against your host** for injecting a few frames of this class; have the program print the IP source and destination addresses and the IP packet's protocol field. Assuming no other traffic is being sent/received by your host, **which is the most frequent number contained in the IP packet's Protocol field**, in this specific case in which we are swamping your host with pings?

5. Investigate whether **sending is possible with libpcap**? If so, **explain** how it can be done?

Appendix 1: Source code of NGN-Lab-Libpcap--test2.c

```
/*
 * Course on NGN @ University of León
 * Dept. IESA (Systems and Electrical Eng)
 * (C) 2015, José María Foces Morán
 *
 * LABS on the pcap capture library
 * NGN-Lab-Libpcap--test2.c
 * v 1.0 9/March/2015
 *
 * Captures n Ethernet frames (n is provided as first command-line
argument) and
 * for each, it prints the header fields
 *
 */

#include <stdio.h>
#include <stdlib.h>
#include <errno.h>

#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>

#include <pcap.h>

unsigned int frameCount = 0;

/*
 * CRC: Last 4 bytes of frame
 */
void printCRC(u_char *frame) {

    printf("\nCRC: ");
    fflush(stdout);

}

/*
 * frame bytes ( 12 and 13 )
 * DESTINATION MAC ADDRESS
 */
void printEthertype(u_char *frame) {

    //ntohs(frame[12])
    printf("\nEthertype: ");
    fflush(stdout);

}

/*
 * Print the MAC addresses
 */
void printMac(u_char *frame) {

    int i;

    printf("Destination MAC: ");
    /*
     * First six bytes of frame ( 0 - 5 )
     * DESTINATION MAC ADDRESS
     */
    for (i = 0; i < 6; i++) {
        printf("%02x:", frame[i]);
    }

    printf("\nSource MAC: ");

    /*
     * Ensuing six bytes of frame ( 6 - 11 )
     * SOURCE MAC ADDRESS
     */
    for (; i < 12; i++) {
        printf("%02x:", frame[i]);
    }

}

/*
 * pcap_pkthdr Generic per-packet information, as supplied by libpcap:
 *      packet lengths and the packet timestamp
 *
 * u_char* frame: Bytes of data from the frame itself with all
 *                the wire bits stripped
 */
void printFrame(const struct pcap_pkthdr *frameHeader, u_char* frame) {

    printf("Header length: %u\n", frameHeader->len);
    printMac(frame);
    printEthertype(frame);
    printCRC(frame);
}

/*
 * Callback function specified into  pcap_loop(...)
 * This callback will capture 1 frame whose header is available in
frameHeader
 * The frame itself is stored into frame
 */
void getNewFrame(u_char *dummy, const struct pcap_pkthdr *frameHeader,
u_char *frame) {
    printf("Packet captured no. %u  ", frameCount++);

    /*
     * Print the frame just captured
     */
    printFrame(frameHeader, frame);

    fflush(stdout);

}

/*
 * printIPandMask(char *defaultDev)
 *
 * Prints the IP address and the Network mask configured into the network
 * device whose p_cap name is into defatultDevice
 *
 */
void printIPandMask(char *defaultDev) {
    bpf_u_int32 netAddress;
    bpf_u_int32 netMask;
    struct in_addr inAddress;
    char errbuf[PCAP_ERRBUF_SIZE];

    printf("Network device name = %s\n", defaultDev);

    /*
     * pcap_lookupnet() returns the IP and the netmask of the passed
device
     * Actual parameters netAddress and netMask are passed by reference
since
     * we want them to hold the IP and the netmask, they are therefore
output
     * parameters
     */
    if (pcap_lookupnet(defaultDev, &netAddress, &netMask, errbuf) == -1)
{
        printf("%s\n", errbuf);
        exit(3);
    }

    /*
     * inet_ntoa() turns a "binary network address into an ascii string"
     */
    inAddress.s_addr = netAddress;
    char *ip;

    if ((ip = inet_ntoa(inAddress)) == NULL) {
        perror("inet_ntoa");
        exit(4);
    }

    printf("IP address = %s\n", ip);

    inAddress.s_addr = netMask;
    char *mask = inet_ntoa(inAddress);

    if (mask == NULL) {
        perror("inet_ntoa");
        exit(5);
    }

    printf("Network mask = %s\n", mask);
    fflush(stdout);
}

unsigned int performCapture(unsigned int nFramesToCapture) {

    char *defaultNetDevice;
    char errbuf[PCAP_ERRBUF_SIZE]; //The pcap error string buffer

    /*
```

```
     * Lookup the default network device on which to capture by invoking
     * pcap_lookupdev()
     */
    defaultNetDevice = pcap_lookupdev(errbuf);

    if (defaultNetDevice == (char *) NULL) {
        printf("%s\n", errbuf);
        exit(2);
    }

    /*
     * Printout of IP address + Net mask
     */
    printIPandMask(defaultNetDevice);

    /*
     * Open network device for capturing frames not-in-promiscuous mode:
     *
     * pcap_t *pcap_open_live(
     * const char *device,
     * int snaplen,
     * int promisc,
     * int timeout_ms,
     * char *errbuf);
     *
     */
    pcap_t* status;
    status = pcap_open_live(defaultNetDevice, BUFSIZ, 0, -1, errbuf);


    if (status == (pcap_t *) NULL) {
        printf("Call to pcap_open_live() returned error: %s\n", errbuf);
        exit(4);
    }

    printf("\n\nCapturing %u frames:\n", nFramesToCapture);
    fflush(stdout);

    /*
     * int pcap_loop(
     * pcap_t *status,
     * int number_of_frames_to_capture,
     * pcap_handler callback_function,
     * u_char *user
     * )
     *
     */
    pcap_loop(status, nFramesToCapture, getNewFrame, (const char *)
NULL);

    return nFramesToCapture;
}

int main(int argc, char *args[]) {
    /*
     * Process command line arguments:
     * get the number of frames to capture
     */
    if (argc != 2) {
        printf("%s <n_frames_to_capture>\n", args[0]);
        exit(-1);
    }

    int frameCount = performCapture(atoi(args[1]));

    printf("\n\nFinished. %u frames captured.\n", frameCount);

}
```