v 1.3

Universidad de León
Bachelor Degree on Computer Science and Engineering
Course on Computer Networks

CN Practice on PF_PACKET Socket Programming

Study Guide

This study guide outline is not to be included in your LabBook writeup

1. Have the textbook by Peterson & Davie at hand. Most of the material that we have taught so far belongs in book chapters 1 and 2. Find the 6th edition to the book, here:

   https://github.com/SystemsApproach/book/releases/download/v6.1/book.pdf

2. A valuable resource as you undertake the practice exercises is the Practices and Questionnaires that we did the past academic year:

   http://paloalto.unileon.es/cn/

3. Programs that access the network or datalink layers directly require a Linux capability known as *Raw Socket Capability*. This capability is usually limited to the system administrator (The root user), but you need it so that the programs that you make can successfully open and read/write onto the raw socket successfully. I have started a service in paloalto.unileon.es that grants the CAP_NET_RAW capability to your programs such that they successfully open a raw socket even if the running user is not the root user. Assuming that your user name is student0 and that your executable file name is magic, the following command will achieve granting magic the CAP_NET_RAW capability:

   ```
   $ echo /home/student0/magic > /home/administrator/fifo.cn
   ```

   Be attentive to send the full path name of your file to fifo.cn, otherwise, your program will not be located by the capability-granting process. A few seconds afterwards, if you check whether *magic* has the CAP_NET_RAW capability, you'll observe that it does have it:

   ```
   $ setcap -v 'CAP_NET_RAW=epi' magic
   magic: OK
   ```

Refresher about Internet Architecture and Interfaces.

In chapter 1, we introduced the Internet Architecture (IA). The essential characteristic of IA is that it is comprised of a hierarchy of four protocol layers alongside interfaces for programmatic access to layers 1, 2 and 3. Each interface permits programs to use the services provided by the accessed layer's protocols. Programming consists of invoking send()[1] or receive() function call from the sockets library. The rest of the communication processes triggered across the TCP/IP protocol stack result completely transparent to you, the programmer; those *processes* are the exclusive responsibility of the socket interface alongside with the underlying protocol stack.

Each of the lower three layers in the IA offers a socket interface, consequently you can make network programs that choose a socket interface to access one of the following layers:

- **Interface to Layer 3**. These sockets allow programming against the UDP and TCP protocols. If time permits, we will introduce these sockets in the last WH while on the distance teaching mode. This interface is commonly named after its designers, the University of California at Berkeley (Berkeley Sockets).

- **Interface to Layer 2**. Basically, these sockets allow a programmer to program against the IP protocol (At the Network layer). The technical name to these sockets is Raw Sockets and are used by notable utilities such as ping and traceroute, with which we have familiarity from Practice 1. The protocol family of this type of raw sockets is PF_INET (Protocol Family_Internet).

- **Interface to Layer 1**. At the layer 1 we find a protocol essential to this course: the Ethernet protocol. The type of socket to use is Raw Sockets again, however, in this case, the protocol family is PF_PACKET (Protocol Family_Packet; this protocol family is exclusive of Linux. The name "Packet" might result somewhat misleading. This is the type of socket that is *the focus of this practice*. Find in fig. 1 the location of PF_PACKET Raw Socket interface in the TCP/IP Architecture.

[1] send() and receive(), in this context are to be understood as representing groups of library functions for sending and for receiving and not concrete socket library functions. The Linux command `$ man send` will provide all the relevant information.

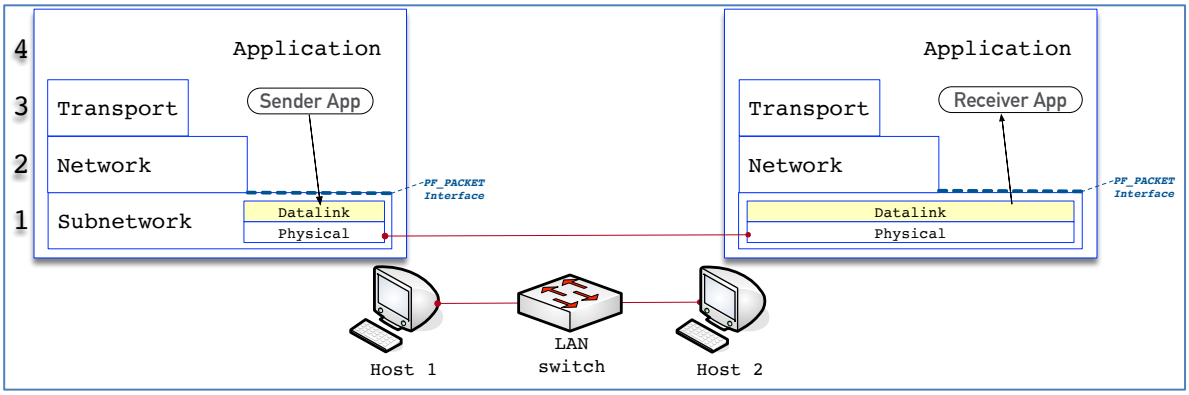

Figure 1. Sender application on host 1 uses the PF_PACKET socket interface to the Ethernet datalink protocol to send information to its peer application on host 2.

Programming against the PF_PACKET socket interface

In this practice we develop a program that sends a frame directly over an Ethernet datalink (In the specific case of this practice, the network adaptor used is eno1). The program transmits the frame by calling the send() function upon a PF_PACKET/Datagram socket. The payload encapsulated into the frame results from repeatedly copying a string of bytes provided by the user on the command line. The transmission is checked by having tcpdump receive the frame, both in the same host that transmits the frame and at the host that owns the Unicast MAC address that the frame is to be sent to. In case the frame is sent to the Broadcast address, the frame should be received by all the hosts within the broadcast domain (LAN segment) where the frame was sent over.

The source code includes comments to the most important data structures and program operations. If necessary, scan the code for a more detailed study thereof. Below, an overall explanation of the program is developed by focusing on the few functions that comprise it and on the socket calls and sockets data structures involved.

The purpose of the proposed exercises is twofold. On the one hand, they are aimed at improving the basis of your knowledge about network architecture and sockets; on the other hand, they aim at improving your hands-on ability in C programming and in Linux networking.

- The program starts with interpreting the command line arguments in function **main()**: the network interface to use for transmission, the base character string that is to be repeatedly copied into the payload and the optional total payload size, which default value is 128B.

- Function **start()** contains the core program operations:

  i) Print an informational program legend

ii) Create a socket address structure for the raw socket that is to be created afterwards.
iii) Create a socket of the right type for sending directly onto an ethernet interface
iv) Create an array for storing the bytes that will make up the payload which is to be encapsulated into the frame to be transmitted
v) Call the socket's function sendto() which will request the sockets layer to send the frame just built

- Sending and receiving on raw sockets entails a socket structure to store key fields from the ethernet header and other relevant information such as the network interface involved in either operation. The type of sockets used in this practice (PF_PACKET family) employ a socket address supported by the following derived type:

  struct sockaddr_ll

  The postfix "ll" in the struct type name means *link level*. This struct is declared in the following header file:

  #include <linux/if_packet.h>

  The creation of the struct sockaddr_ll necessary for sending and receiving is accomplished in function **fillSocketAddress()**. The function fills the necessary fields as expanded on the following outline:

  - The address family (PF_PACKET)
  - The interface index (ifconfig returns an interface name which must be translated into an index valid for the socket address)
  - Ethernet's multiplexing key (Ethertype) to be used by this socket (Note that the selected Ethertype must be stored in *network byte order)*. We use an Ethertype value that is not reserved (0x07ff).
  - The remaining fields are related with ARP (We take up ARP in a forthcoming Lecture and Practice). You can see their defaults in the source code.

  This function returns the struct sockaddr_ll filled with the values explained above. An essential value remains yet to be filled: the Destination MAC address which is set by function setDestinationMAC() initially to the Broadcast address. Later, an exercise will ask you to change that address to the Unicast MAC address of a concrete, known host within the Lab's LAN.

- The PF_PACKET socket (Of type SOCK_DGRAM) is created in line 105:
  - The socket family argument is constant PF_PACKET
  - The socket communication style if SOCK_DGRAM, this means that the programmer only provides the payload to be sent. The Ethernet header will be built by the sockets library.

- The last argument is the Ethertype field value (The same value that we created above for the socket address)

- The payload to be included in the frame to be sent over Ethernet is created in function **buildPayload()**. This function appends a number of copies of a string provided by the user on the command line. The total number of bytes to be included in the payload is given in the third command line argument and has a default value of 128 Bytes and a maximum given by the default Ethernet MTU of 1500 Bytes.

Sundry technical information

MAC addresses of relevant Lab B6's host interfaces
`http://paloalto.unileon.es/cn/Q/mac-ip.txt`

Source code
http://paloalto.unileon.es/cn/Q/dgramPFPACKETSend.c

Exercises for practice

1. Checking the base program:

a. The source code is contained in file `dgramPFPACKETSend.c`. Download the program and compile it:

```
$ wget paloalto.unileon.es/cn/Q/dgramPFPACKETSend.c
$ gcc -o dgramPFPACKETSend dgramPFPACKETSend.c
```

b. Check that the program runs; provide no command line arguments at this time. Notice the information provided by the program about the expected command line syntax.

c. Send the program to the administrator's FIFO to have its CAP_NET_RAW capability set so that the program later runs ok without complaining about your lacking the necessary privileges.

```
$ echo /home/<your user name>/dgramPFPACKETSend > /home/administrator/fifo.cn
```

d. Now, open a new session in paloalto.unileon.es with ssh -p 50500. Use the new session for connecting with another host in the Lab's network. Power it up if necessary by sending it the magic packet (Lookup the host adapter's MAC in the table above in section titled Sundry Technical Information). After waiting for about a couple minutes, issue an ssh to the chosen host form the second Terminal

Window and log on with user *administrator* and password '*19xxdpq16*'. When you are in session, type an appropriate tcpdump command line that will receive the frame sent by program dgramPFPACKETSend. You may consult the past practice for an example of tcpdump command line arguments appropriate for this exercise, one similar to this one:

```
$ tcpdump -i eno1 -e -XX -vvv ether proto 0x07ff
```

Note that you will have to execute tcpdump with sudo or by first switching to root user with the su command.

e. While tcpdump is waiting to receive the expected frames, start the dgramPFPACKETSend program and pass it the necessary command line arguments:

- interface name
- A base string like 'Andra Tutto Bene'
- The total payload size to send

Notice: *The following exercises entail that you modify the base program. You can modify the source code file dgramPFPACKETSend.c at the trampoline host in Network 2 by using the vi editor (Or vim) or maybe the nano editor; alternately, you can do the modifications locally in your home computer with your preferred editor, and then have the source code uploaded later to your account in paloalto at every increment and then have it recompiled (Every time you modify the source code, either remotely or locally, you'll have to recompile it).*

2. Modify the program so that it accepts the Ethertype from the command line as a 4 hex-digit string. This argument is not optional.

a. Create a few unit tests to check that your program functions as expected. Monitor your program's operation with tcpdump at another host in the Lab B6 network.

3. Modify the program so that it accepts the Destination MAC address from the command line in hexadecimal base and separating each byte with a colon (The same MAC address format used by ifconfig). This argument is not optional.

a. Create a few unit tests to check that your program functions as expected. Monitor your program's operation with tcpdump at another host in RemLabB6 by previously opening a new ssh session with it.

Program source code

```
/*
 * Conceptual Computer Networks textbook
 * CN course 2020
 * Express Practice: Simple Datagram PF_PACKET DGRAM send program
 * dgramRawSend.c
 * All rights reserved:
 * (C) 2020 by José María Foces Morán & José María Foces Vivancos
 */

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

#include <fcntl.h>
#include <memory.h>

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

#include <linux/if_ether.h>
#include <linux/if_packet.h>
#include <net/ethernet.h>
#include <net/if.h>

#include <signal.h>
#include <errno.h>
#include <sys/time.h>

#define byte u_char
#define TRUE 1
#define ETHERTYPE_EXPERIMENTAL 0x07ff
#define DEFAULT_MTU 1500

void setDestinationMAC(byte *p) {

    /*
     * Fill the byte array pointed to by variable p with the
     * bytes that comprise the broadcast address (All 1's)
     */
    p[0] = (byte) 0xff;
    p[1] = (byte) 0xff;
    p[2] = (byte) 0xff;
    p[3] = (byte) 0xff;
    p[4] = (byte) 0xff;
    p[5] = (byte) 0xff;

}

/*
 * Create a "large" payload of size payloadSize by cloning baseDataToClone[] as
 * many times as it fits including the remainder size
 */
char * buildPayload(char *baseDataToClone, unsigned baseDataSize, unsigned
payloadSize) {
```



```
    char *p = (char *) '\0';

    if (payloadSize == 0 || baseDataSize == 0) {
        fprintf(stderr, "Error: Requested payload size and base data size must be both
greater than 0.\n");
        return p;
    }

    if (payloadSize > DEFAULT_MTU) {
        fprintf(stderr, "Error: Requested payload size exceeds Ethernet maximum
MTU.\n");
        return p;
    }

    if (payloadSize < baseDataSize) {
        fprintf(stderr, "Error: Payload size cannot fit the base data to be
cloned.\n");
        return p;
    }

    /*
     * Request dynamic memory space for the payload to be built
     * p is a sentinel marking the start of the payload being built
     * We'll return p to our calling function
     */
    p = malloc(payloadSize);

    /*
     * Copy p into q and use the latter for indexing the payload as
     * we fill it in the for loop below
     */
    char *q = p;

    /*
     * Copy baseDataToClone as many times as it fits payloadSize
     * Each copy of baseDataSize bytes is made by calling memcpy()
     */
    for (int i = 0; i < (payloadSize / baseDataSize); i++) {
        memcpy(q, baseDataToClone, baseDataSize);
        //Move q pointer forward baseDataSize bytes
        q += baseDataSize;
    }

    /* If the integer division (payloadSize / baseDataSize) produces a
     * remainder (payloadSize % baseDataSize), copy the number of bytes
     * represented by the remainder from baseDataToClone to q
     */
    memcpy(q, baseDataToClone, payloadSize % baseDataSize);

    return p;
}

void printProgramLegend(char *payload) {

    printf("Send a frame with PF_PACKET/SOCK_DGRAM\n");
    printf("\tDMAC = ff:ff:ff:ff:ff:ff\n");
    printf("\tSMAC = Network Interface's MAC\n");
    printf("\tEtherytpe = %hx\n", ETHERTYPE_EXPERIMENTAL);
    //printf("\tPayload=\"%s\"", payload);

    fflush(stdout);
```



```
}

/*
 * This function fills the fields of the socket address structure
 * Some of the come from the command line arguments passed by the user
 *
 * u_int16_t is  used for representing the ethertype
 * u_int16_t is declared int /usr/include/x86_64-linux-gnu/sys/types.h
 * with #include <sys/types.h>
 *
 * __be16 is defined in /usr/include/linux/types.h and is also used for
 * representing ethernet's ethertype field
 */
struct sockaddr_ll fillSocketAddress(char *ifName, u_int16_t ethertype) {

    /*
     * sockaddr_ll has slightly different used when sending than are used when
     * receiving.
     * When sending, sockaddr_ll stores the Destination MAC and the
     * multiplexing key (Ethertype) and the index of the interface to be used for
     * actually transmitting the frame
     *
     * When receiving, sockaddr_ll stores the Source MAC address, the received
     * Ethertype and the interface index the frame was received onto
     */
    struct sockaddr_ll socketAddress;

    //T
    socketAddress.sll_family = PF_PACKET;

    /* Index of network interface */
    socketAddress.sll_ifindex = if_nametoindex(ifName);
    if (socketAddress.sll_ifindex == 0) {
        perror("Error indexing interface name");
        exit(-2);
    }

    /* Address length*/
    socketAddress.sll_halen = ETH_ALEN;

    //Ethertype translated to Network Byte Order
    socketAddress.sll_protocol = htons(ethertype);

    //arp-related
    socketAddress.sll_hatype = 0;

    //arp-related
    socketAddress.sll_pkttype = 0;

    return socketAddress;

}

void start(char *ifName, char *baseDataToClone, int baseDataSize, int payloadSize) {

    /*
     * Print the frame fields when program begins to run
     */
    printProgramLegend(baseDataToClone);

    /*
     * This struct stores basic Raw socket parameters such as:
     * multiplexing key (Ethertype), Destination MAC, etc
```



```
     */
    struct sockaddr_ll socketAddress = fillSocketAddress(ifName, (u_short)
ETHERTYPE_EXPERIMENTAL);
    setDestinationMAC(&(socketAddress.sll_addr[0]));

    /*
     * Create a socket with the following three actual parameter values:
     *
     * Arg 1: PF_PACKET is the address family used by Ethernet/Datalink sockets in
Linux
     *
     * Arg 2: SOCK_DGRAM is the type of communication style to be used with this
socket:
     *  - SOCK_DGRAM means that the programmer is letting the building of the
     *    frame's header to the sockets layer (The service interface itself),
     *    i.e., the programmer is not building the header but is only the payload
     *    as we did above.
     *
     *  - The other option available for this argument is constant SOCK_RAW
     *    which means that the programmer is providing a full datalink header
     *
     * Arg 3: Ethernet's multiplexing key (Ethertype); in this case we are using
     *  - ETHERTYPE_EXPERIMENTAL (0x07ff) which is not reserved. Since we already
     *    loaded value ETHERTYPE_EXPERIMENTAL into the socket address created above,
     *    we use it again, for consistency (socketAddress.sll_protocol)
     *
     */
    int sock = socket(PF_PACKET, SOCK_DGRAM, socketAddress.sll_protocol);

    /*
     * Call function to have the payload built from a base array of bytes that is
     * going to be cloned a number of times. baseDataToClone is entered by the user
     * on the command line. payloadSize is the total size of the payload
     */
    byte *p = buildPayload(baseDataToClone, baseDataSize, payloadSize);
    if (p == (byte *) '\0') {
        exit(-1);
    }

    /*
     * Finally, the data is ready to be sent onto socket sock
     * sock:    The socket created above
     * p:       Pointer to an array of bytes (unsigned char) that contains the
     *          data to be sent onto the socket
     * payloadSize:
     *          The size of the array pointed to by p in bytes
     * 0:       Options for this socket
     * socketAddress:
     *          Storage for the socket address which contains fields such as:
     *              · Dest MAC address
     *              · Ethertype (Ethernet's multiplexing key)
     *              · Consult /usr/include/linux
     */
    if (sendto(sock, p, payloadSize, 0, (struct sockaddr *) &socketAddress, sizeof
(socketAddress)) == -1) {
        printf("\nsendto() call failed\n");
        perror("sendto: ");
        exit(-1);
    }
}

int main(int argc, char** argv) {
```



```
    /*
     * Command-line processing
     */
    if (argc == 3) {

        //call start with default size of 128B
        start(argv[1], argv[2], strlen(argv[2]), 128);
        printf("Simple frame successfully sent to the broadcast address via %s.\n",
argv[1]);

    } else if (argc == 4) {

        int paySize = atoi(argv[3]);
        if (paySize < 0 || paySize > DEFAULT_MTU) {
            fprintf(stderr, "Invalid value for Payload size to send: %d; should be [1,
1500]\n", paySize);
            exit(-1);
        }
        //call start by passing no default parameter
        start(argv[1], argv[2], strlen(argv[2]), paySize);
        printf("Simple frame successfully sent to the broadcast address via %s.\n",
argv[1]);

    } else {

        fprintf(stderr, "Usage: %s\n", argv[0]);
        fprintf(stderr, "\t<Network Interface>\n");
        fprintf(stderr, "\t<Data to be cloned between quotation marks (\"...\")>\n");
        fprintf(stderr, "\t[Payload size to send (Max. 1500B, Default 128B)]\n");

        exit(-1);
    }

}
```

