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

CN Practical on PF_PACKET Socket Programming

Refresher about Internet Architecture and Interfaces.

In chapter 1, we introduced the Internet Architecture (IA). 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 protocol's interface permits programs to use the services provided by it. The same Linux system call is invoked in all cases, its name is socket(), by passing it a parameter list specific to each different protocol layer. In short, after socket() has been called, all of the programming consists of invoking some form of the send()[1] and recv() functions. The resulting communication processes triggered thereafter across the TCP/IP protocol stack are completely transparent to you, the programmer; those *processes* are the exclusive responsibility of the socket interface alongside 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 any socket interface to access one of the following layers:

- **Interface to Layer 3**. These sockets allow programming against the UDP and TCP protocols. We will introduce these sockets in the last practical of this academic year. 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, *i.e.* 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 the practicals done so far. The protocol family of this type of Raw Sockets is PF_INET (Protocol Family_Internet).

- **Interface to Layer 1**. At the layer 1 (The Subnetwork layer) we find a protocol essential to this course: the Ethernet protocol. The type of socket to

[1] send() and recv(), 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 about functions send(), sendto() and sendmsg(); in the case of recv(), command `$ man recv` will printout the details about function calls recv(), recvfrom() and recvmsg().

use is Raw Sockets again, however, in this case, the protocol family is PF_PACKET. Furthermore, we should note by now that protocol family PF_PACKET is exclusive of Linux. The name "Packet" might result somewhat misleading, though, for the packet PDU is usually applied to network protocols, not to subnetwork protocols. Despite all of this fuss over the name packet, we must be aware of its significance in layer-1 socket programming. This is the type of socket that is *the focus of this practical*. Find in fig. 1 the location of PF_PACKET Raw Socket interface in the TCP/IP Architecture.

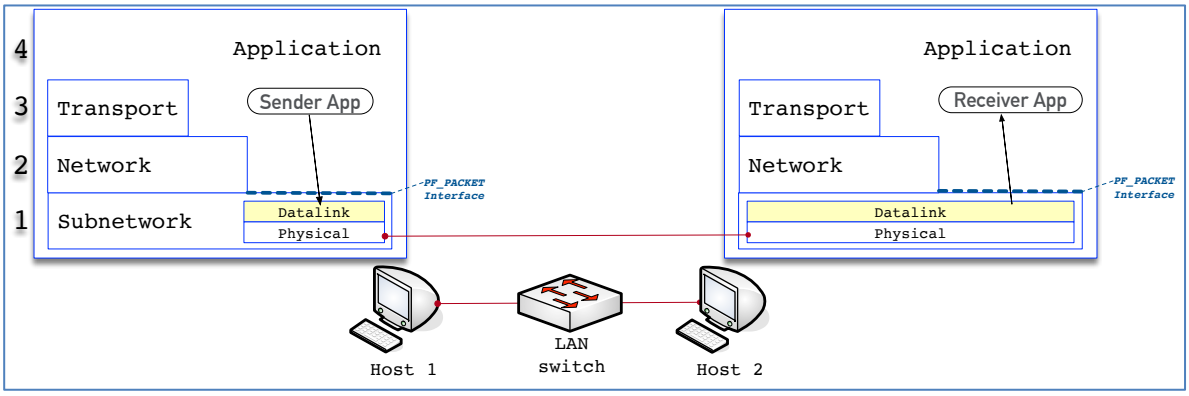

Figure 1. Sender application on host 1 uses the PF_PACKET socket interface to the Ethernet datalink protocol to send information to its peer application on host 2.

Overview of the sample layer-1 socket program

In this practice we are to develop a program that sends a frame directly over an Ethernet datalink (In the specific case of this practical, the network adaptor's interface name is eno1; your system may use a different one). The program sends the frame by calling one of send() functions available in Linux (sendto() and other derivatives) 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 user must provide the receiving host's interface's MAC address on the command line. The sending of the frame is checked by using tcpdump on the sending side, and on the receiving side as well. In case the the frame is sent to the Broadcast address (Recall that that MAC address is comprised of 48 bits 1, or 12 hex f characters) the frame should be received by all the hosts within the broadcast domain (LAN segment) where the sending host is connected.

The source code provided includes comments to the most important data structures and program operations. If necessary, scan the code for a more detailed study. 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.

Following the program explanations that come next, a number of exercises are included. Some of them must be done in the Lab, some others are meant to be completed in your study time. The purpose of the proposed exercises is twofold. On the one hand, they are aimed at expanding your network programming ability; on the other hand, they are aimed at improving your hands-on ability in administering and troubleshooting Linux networking.

- In function **main()** the program starts with *parsing* the command line arguments:
    - The network interface to use for sending the frame
    - The base character string that is to be repeatedly copied into the payload
    - The optional total payload size, which default value is 128B and which maximum is 1500B.
    - Function main exits altogether, or calls **start()** for further processing.

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

    1. Print an informational program legend
    2. Create a socket address structure for the raw socket that is to be created afterwards.
    3. Create a socket of the right type for sending directly onto an ethernet interface
    4. Create an array for storing the bytes that will make up the payload which is to be encapsulated into the frame to be transmitted
    5. Call the socket's function sendto() which will request the sockets layer(Sockets interface) 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 Linux network interface involved in both operations. The type of sockets used in this practical (PF_PACKET family) employ a socket address supported by the following C struct type:

    ```
    struct sockaddr_ll …
    ```

    The postfix "ll" in the struct type name means *link level*. This struct is

declared in the following header file, so if you include it, you need not declare it before you use it for storing a PF_PACKET socket address:

```
#include <linux/if_packet.h>
```

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

- The address family (PF_PACKET)
- The interface index. The ifconfig command returns an *interface name* which, for our program to function ok must be translated into an *interface index*. The program computes the interface index by calling if_nametoindex(). The latter function returns the interface index, which can subsequently be passed on to a call to function getifaddrs(3) in order to obtain a number of attirbutes to the network interface, such as its IP address, its Netmask, etc.
- 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) in the IEEE standard that sets MAC address use.
- 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 209:
  - The socket family argument is constant PF_PACKET, required for Subnetwork-layer (Ethernet) sockets in Linux
  - The socket communication style if SOCK_DGRAM. This means that the programmer only provides the payload to be sent onto the underlying Ethernet adaptor. The Ethernet header will be built by the sockets interface itself. A communication style of SOCK_RAW can be set on the socket instead of SOCK_DGRAM which will require the programmer to include all of the Ethernet frame's fields:

        - Destination MAC
        - Source MAC
        - Ethertype
        - Payload

    - 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.

Exercise 1 [Lab]

i. Download the base source code from the correct URL depending on whether you are connected to LAB B6 network directly or via the Internet, respectively:

http://192.168.1.89/cn/labs/dgramPFPACKETSendv2.c

http://paloalto.unileon.es/cn/labs/dgramPFPACKETSendv2.c

You can download the file by using the wget command-line utility or a Web browser:

```
$ wget paloalto.unileon.es/cn/labs/dgramPFPACKETSendv2.c

$ wget 192.168.1.89/cn/labs/dgramPFPACKETSendv2.c
```

ii. Compile it:

```
$ gcc -o dgramPFPACKETSendv2 dgramPFPACKETSendv2.c
```

iii. Switch to super-user (su), then check that the program runs correctly; provide no command line arguments at this time. Notice the information provided by the program about the expected command line syntax.

```
[Term 1] $ su
…
[Term 1] # ./dgramPFPACKETSendv2
          <Network Interface>
          <Data to be cloned between quotation marks ("...")>
          [Payload size to send (Max. 1500B, Default 128B)]
```


iv. Run tcpdump on a new terminal window with options appropriate for capturing the frame that the `dgramPFPACKETSendv2` program is to send below in step v.

```
[Term 2] $ su
…
[Term 2] # tcpdump -i eno1 -eXX -vvv ether proto 0x07ff
```

v. While tcpdump is waiting to watch the frame that the program is about to send, start the `dgramPFPACKETSendv2` program and pass it the necessary command line arguments:

- Interface name (**eno1**, for example)
- A base string like 'Andra Tutto Bene'
- The total payload size to send (Max. 1500 Bytes, or 128 Bytes by default)

vi. Explain the contents of following Ethernet frame fields captured by tcpdump:

- Destination MAC. Is this the **broadcast MAC** address? Explain its semantics.

- Source MAC. Is this the MAC address of the interface name that you passed on to the program as an argument? Check it with ip or ifconfig.

- Ethertype. Is the 16-bit Ethertype value the same that is used and produced by the program?

Exercise 2 [Lab]

i. Now, we want to check sending the same frame to a Unicast MAC address. To that end, obtain the MAC address of one of the Lab's hosts and enter that MAC address in the function where the destination MAC address is set. Recompile the `dgramPFPACKETSendv2.c` program.

ii. Run an instance of tcpdump in the destination host. Provide arguments that are correct for this purpose. Include below images or texts that show the results you have obtained. Opening a remote ssh session from your current computer may facilitate the task ($ ssh administrator@192.168.1.<N>).

iii. Run an instance of tcpdump in your computer.

iv. Now, run the modified `dgramPFPACKETSendv2 program`.

v. Explain whether the **order** in which you enter the bytes in the address byte array is significant. Is it **significant**? Include below images or texts that show the results you have obtained. According to the conclusions you have arrived at in the preceding question, is the byte ordering Little Endian, or Byte Endian?

Exercise 3 [Home]

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

ii. 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.

iii. 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.

dgramPFPACKETSendv2.c source code

```
/*
 * Conceptual Computer Networks textbook
 * CN course 2025
 * Express Practice: Simple Datagram PF_PACKET DGRAM send program
 * dgramPFPACKETSend.c
 * All rights reserved:
 * (C) 2022-2025 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 MAC address of the LAN's host
     * you intend to send some frame to (At the moment, that MAC address is
     * the BROADCAST MAC address)
     */

    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 (char *ifName, u_int16_t ethertype) {

    /*
     * sockaddr_ll is used differently when sending than it is 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 in filling up the socket address' field:
     *    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 (For the time, leave as is, 0)
     * 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);
    }

}
```

