Vysoké učení technické v Brně
Fakulta informačních technologií

IPK−project no. II

Packet Sniffer (ZETA variant)

April 24, 2022

Patrik Korytár (xkoryt04)

Contents

1 Packet Sniffer Theory

1.1 TCP/IP stack

TCP/IP stack is a set of protocols typically used across Internet and other computer networks [8].

TCP/IP stack consists of multiple layers and these layers provide abstraction on user's data that need to be delivered and data that help deliver the user's data. The highest data includes just user's data and this provides highest abstraction on how the data are actually delivered. Lower layers also provide headers and mechanisms that enable user's data to be actually delivered.

TCP/IP consists of the following layers. Application layer just contains user's application data. It includes no data on how to deliver the data. Transport layer includes port addressing to deliver the user's data between applications. Internet layer by the use of IP addresses delivers user's data across networks [7]. Link layer delivers user's data across one network [9]. Finally, physical layer provides no abstraction whatsoever and just describes how the data are transmitted on the physical medium.

Eventually, only bytes of data are delivered across network. All networking devices analyze these bytes, and based on the layer they operate on and protocols that are in use, they get various information.

1.2 Packet Sniffer

The goal of a packet sniffer is to analyze and print information about packets delivered on specified interface [10]. Packet sniffer reads consecutive bytes and gets data from headers that are encapsulated one in another. These headers include addresses (MAC or IP, for example), information needed by the protocol (ICMP type, for example) and identification of next encapsulated protocol header. By knowing what protocol header follows, the program knows how to interpret the data. All or some of data from headers are printed, sometimes including printing of whole packet in hexadecimal and/or ASCII format.

1.3 Interface traffic capture

For the program to be able to get access to all packets that are delivered on the interface, the promiscuous mode must be used. Promiscuous mode causes all packets delivered on the network interface card to be passed to the CPU [11].

2 How to use the program

Description on how to run the program, list of parameters and their usage is described in the README.md file.

3 Technologies used

Implementation uses C++17 language and its standard library. Program *make* is used to run the compilation (or manual compilation can be used).

3.1 Used libraries

Other than standard library, program uses functionality from *pcap* library to capture packets on interfaces. POSIX library and its header files arpa/inet.h and netinet/*.h are included to get access to data structures for parsing the packets. signal.h header is also included to handle the SIGINT signal (ctrl+Z).

3.2 Code documentation

Code is documented using Doxygen comments. Please refer to the code and commented functions to get more information about the implementation. High-level description is included below.

4 Implementation outline

4.1 Code structure

main.cpp file is the main entrypoint of the whole program. Other functionality is implemented inside *src* directory.

Code is divided into following modules:

- utils – CLI parameters and error handling
- sniffer – creation of sniffer and sniffer filter
- frame_parser – parsing frame contents

4.2 utils module

To get program's parameters, I implemented my own parameter parser. It consists of for cycle and bunch of if statements to check if parameter is valid. If parameter requires certain argument (e.g., port number), the argument existence and value is checked and extracted (possibly converted to number). When and invalid parameter or parameter's argument is entered, error is printed to STDERR and program is exited with an error code.

All information about parameters is collected inside Arguments structure (port number, protocols, etc.). It also has boolean property 'all' to store that no protocol parameter was entered.

4.3 sniffer module

To get and filter packets from interface, pcap library and network interface card's promiscuous mode are used.

All informations about pcap functions I learned from the manual pages on TCPDUMP page [3]. I also read 'Programming with pcap' article to learn how to use pcap [4]. Some useful information about POSIX libraries I learned from tutorial on Stanford University webpage [12].

If the *--interface* parameter is not entered or missing an argument, all interfaces are printed to STDOUT (function *print_interfaces* in this module). To get all interfaces on the device, *pcap_findalldevs* function is used.

If *--interface* parameter is entered with interface specified, sniffer starts sniffing (function *sniff* in this module).
pcap_open_live function opens interface in promiscuous mode. *pcap_datalink* function checks whether specified link layer procol is used on the interface (in this case, Ethernet). *pcap_loop* function gets packets from interface and calls callback function for each packet that is delivered. This function also allows to specify number of packets to be processed.

Program also uses pcap filter capability to filter packets that are displayed (functions set_filter and filter_string in this module). Using parameters from CLI, I constructed filter string for the pcap library. Information about this string can be learned on TCPDUMP page [2]. The string is compiled by the function *pcap_compile.* function *pcap_setfilter* then applies compiled filter string to the interface capture.

4.4 frame_parser module

The data of frames that pcap library gets is stored in an array of bytes. The best aproach to parse the frames was to use pointers to structures and pointer arithmetic, because this is the most flexible one and it allows some sort of dynamic typing (C/C++ does not care what data pointers actually point to).

Needed struct definitions are included inside netinet/*.h header files (for example, netinet/if_ether.h). To get information about headers' structure, I refered to the implementation of these header files on my system.

parse_frame function in this module is used as a callback to the *pcap_loop* function.

Information about frame and captured length and timestamp are inside *pcap_pkthdr* structure passed to the callback. To print timestamp, I used *localtime* and *strftime* standard C functions [5].

Program then goes through headers and prints some informatiom included in these headers. Pointer arithmetic is used to get access to specific headers in the packet and typecasting to the structure to interpret the header data correctly. The underlying encapsulated protocols are learned from specific fields included in the header (this is specific to the protocols). This parsing is included in function *print_ether_info* and other functions of similar name in this module.

To print IPv4 address, I used *inet_ntoa* function. To print IPv6 address, I used *inet_ntop* function. To print MAC address, I implemented my own function *print_mac_addr*

Function *print_frame* in this module prints frame in hexadecimal and ASCII format. It calls other function in this module - *print_frame_ascii*. Implemenation of these consists of mainly for loop and modulo operator. In the ASCII printing, visible characters in the ASCII range of 33 - 126 are printed normally and other characters are printed as dots.

4.5 ARP header

There was a problem with inclusion of header file netinet/if_arp.h. It was not always present in the system, and even if it was, the definition of structure *arphdr* was incomplete. I took inspiration from netinet/if_arp.h [1] and Wikipedia article about ARP [6] and included complete ARP header structure with proper fields in the code (src/frame_parser.h file).

5 Outputted data

For every packet, timestamp, captured frame length and actual frame length is outputted and frame is printed in hexadecimal and ASCII format. For every header, respective protocol is printed (Ethernet, ARP, IPv4, IPv6, ICMP, TCP, UDP).

For Ethernet frame, source and destination MAC address is printed. For ARP, MAC addresses, IP addresses and ARP operation are printed. For IPv4 and IPv6, source and destination IP address is printed along with time to live (hop limit). For ICMP, ICMP type is printed. For TCP and UDP, source and destination port is printed.

6 Testing

For the purpose of testing, I used Wireshark program. I compared output of my program with Wireshark program to check correct behaviour. Subsections below describe creation of needed types of frames/packets.

I tested both on wireless network interface and loopback interface.

6.1 Testing TCP segments

To generate TCP segments, I used my HTTP server implementation from first IPK project (HTTP uses TCP). I started my server and then sent HTTP request with the use of **curl** program.

```
sudo ./ipk-sniffer -i lo --tcp -p 8000 -n 0
curl http://127.0.0.1:8000/hostname
```

6.2 Testing UDP datagrams

To generate UDP datagrams, I used **traceroute** program and scanning of port 53 (DNS service).

```
sudo ./ipk-sniffer -i wlo1 --udp -p 53 -n 0
traceroute youtube.com
```

6.3 Testing ICMP packets

To test ICMP, I used linux command **ping** to ping some IP address.

```
sudo ./ipk-sniffer -i wlo1 --icmp -n 0
ping 8.8.8.8
```

6.4 Testing ARP frames

To test ARP, I also used **ping** command, but this time I pinged local IP address (because then computer tries to learn MAC address of a given IP in the same network).

Given that my network IP is 192.168.1.0/24:

```
sudo ./ipk-sniffer -i wlo1 --arp -n 0
ping 192.168.1.150
```

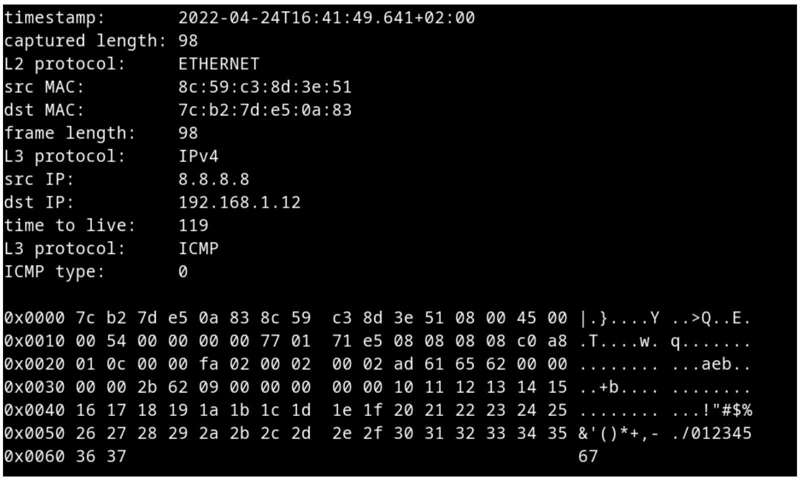
6.5 Testing IPv6 packets

To generate some IPv6 traffic, I used, for example, **ping** command to ping some link-local IPv6 address.

```
sudo ./ipk-sniffer -i wlo1 --icmp -n 0
ping -6 fe80::f265:f862:fbb3:6732%wlo1
```

6.6 Example of testing

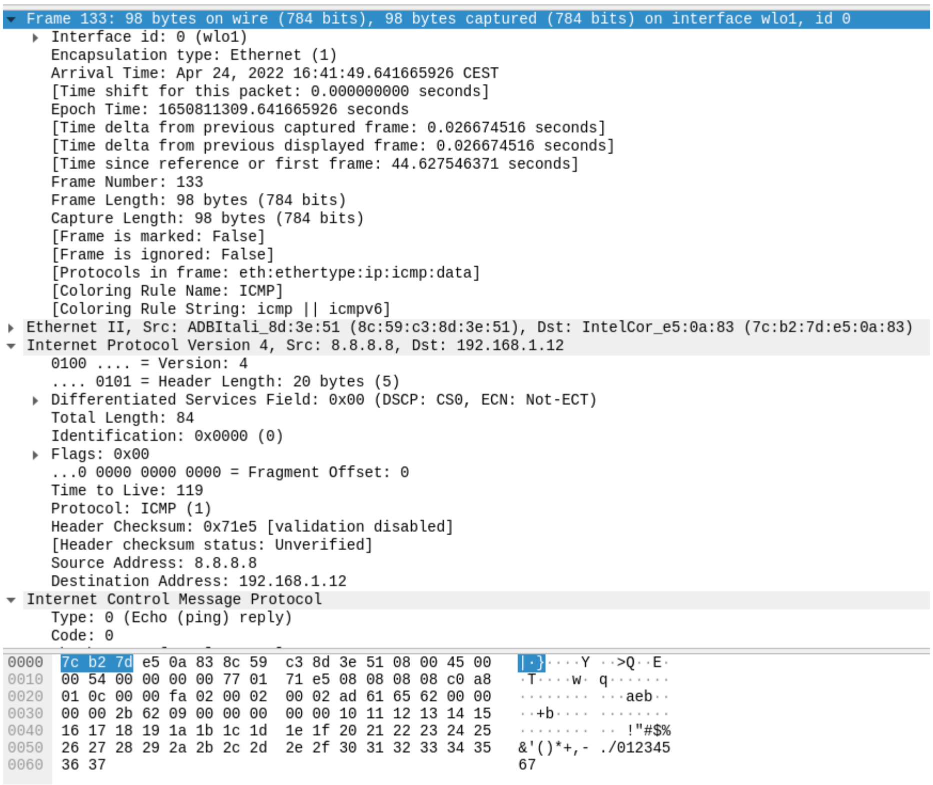
Example of testing ICMP with **ping** command and IP 8.8.8.8. As you can see, output is same.



```
timestamp:       2022-04-24T16:41:49.641+02:00
captured length: 98
L2 protocol:     ETHERNET
src MAC:         8c:59:c3:8d:3e:51
dst MAC:         7c:b2:7d:e5:0a:83
frame length:    98
L3 protocol:     IPv4
src IP:          8.8.8.8
dst IP:          192.168.1.12
time to live:    119
L3 protocol:     ICMP
ICMP type:       0

0x0000 7c b2 7d e5 0a 83 8c 59  c3 8d 3e 51 08 00 45 00 |.}....Y ..>Q..E.
0x0010 00 54 00 00 00 00 77 01  71 e5 08 08 08 08 c0 a8 .T....w. q.......
0x0020 01 0c 00 00 fa 02 00 02  00 02 ad 61 65 62 00 00 ........ ...aeb..
0x0030 00 00 2b 62 09 00 00 00  00 00 10 11 12 13 14 15 ..+b.... ........
0x0040 16 17 18 19 1a 1b 1c 1d  1e 1f 20 21 22 23 24 25 ........ ...!"#$%
0x0050 26 27 28 29 2a 2b 2c 2d  2e 2f 30 31 32 33 34 35 &'()*+,- ./012345
0x0060 36 37                                            67
```

Figure 1: My ICMP ouput



```
Frame 133: 98 bytes on wire (784 bits), 98 bytes captured (784 bits) on interface wlo1, id 0
    Interface id: 0 (wlo1)
    Encapsulation type: Ethernet (1)
    Arrival Time: Apr 24, 2022 16:41:49.641665926 CEST
    [Time shift for this packet: 0.000000000 seconds]
    Epoch Time: 1650811309.641665926 seconds
    [Time delta from previous captured frame: 0.026674516 seconds]
    [Time delta from previous displayed frame: 0.026674516 seconds]
    [Time since reference or first frame: 44.627546371 seconds]
    Frame Number: 133
    Frame Length: 98 bytes (784 bits)
    Capture Length: 98 bytes (784 bits)
    [Frame is marked: False]
    [Frame is ignored: False]
    [Protocols in frame: eth:ethertype:ip:icmp:data]
    [Coloring Rule Name: ICMP]
    [Coloring Rule String: icmp || icmpv6]
Ethernet II, Src: ADBItali_8d:3e:51 (8c:59:c3:8d:3e:51), Dst: IntelCor_e5:0a:83 (7c:b2:7d:e5:0a:83)
Internet Protocol Version 4, Src: 8.8.8.8, Dst: 192.168.1.12
    0100 .... = Version: 4
    .... 0101 = Header Length: 20 bytes (5)
    Differentiated Services Field: 0x00 (DSCP: CS0, ECN: Not-ECT)
    Total Length: 84
    Identification: 0x0000 (0)
    Flags: 0x00
    ...0 0000 0000 0000 = Fragment Offset: 0
    Time to Live: 119
    Protocol: ICMP (1)
    Header Checksum: 0x71e5 [validation disabled]
    [Header checksum status: Unverified]
    Source Address: 8.8.8.8
    Destination Address: 192.168.1.12
Internet Control Message Protocol
    Type: 0 (Echo (ping) reply)
    Code: 0

0000  7c b2 7d e5 0a 83 8c 59  c3 8d 3e 51 08 00 45 00   |·}····Y ··>Q··E·
0010  00 54 00 00 00 00 77 01  71 e5 08 08 08 08 c0 a8   ·T····w· q·······
0020  01 0c 00 00 fa 02 00 02  00 02 ad 61 65 62 00 00   ········ ···aeb··
0030  00 00 2b 62 09 00 00 00  00 00 10 11 12 13 14 15   ··+b···· ········
0040  16 17 18 19 1a 1b 1c 1d  1e 1f 20 21 22 23 24 25   ········ ·· !"#$%
0050  26 27 28 29 2a 2b 2c 2d  2e 2f 30 31 32 33 34 35   &'()*+,- ./012345
0060  36 37                                              67
```

Figure 2: Wireshark ICMP ouput

References

[1] Free Software Foundation, Inc: Definitions for Address Resolution Protocol. [online]. [Accessed: 24/04/2022].
URL `https://sites.uclouvain.be/SystInfo/usr/include/net/if_arp.h.html`

[2] tcpdump: MAN page on TCPDUMP. [online]. Last modified: 17/01/2022 [Accessed: 24/04/2022].
URL `https://www.tcpdump.org/manpages/tcpdump.1.html`

[3] tcpdump: TCPDUMP manpages. [online]. [Accessed: 24/04/2022].
URL `https://www.tcpdump.org/manpages/`

[4] Tim Carstens: Programming with pcap. [online]. [Accessed: 24/04/2022].
URL `https://www.tcpdump.org/pcap.html`

[5] tutorialspoint: C library function - strftime(). [online]. [Accessed: 24/04/2022].
URL `https://www.tutorialspoint.com/c_standard_library/c_function_strftime.htm`

[6] Wikipedia: Address Resolution Protocol. [online]. Last modified: 06/04/2022 [Accessed: 24/04/2022].
URL `https://en.wikipedia.org/wiki/Address_Resolution_Protocol`

[7] Wikipedia: Internet layer. [online]. Last modified: 26/01/2022 [Accessed: 24/04/2022].
URL `https://en.wikipedia.org/wiki/Internet_layer`

[8] Wikipedia: Internet protocol suite. [online]. Last modified: 18/04/2022 [Accessed: 24/04/2022].
URL `https://en.wikipedia.org/wiki/Internet_protocol_suite`

[9] Wikipedia: Link layer. [online]. Last modified: 25/03/2022 [Accessed: 24/04/2022].
URL `https://en.wikipedia.org/wiki/Link_layer`

[10] Wikipedia: Packet analyzer. [online]. Last modified: 10/04/2022 [Accessed: 24/04/2022].
URL `https://en.wikipedia.org/wiki/Packet_analyzer`

[11] Wikipedia: Promiscuous mode. [online]. Last modified: 19/02/2022 [Accessed: 24/04/2022].
URL `https://en.wikipedia.org/wiki/Promiscuous_mode`

[12] yuba.stanford.edu: The Sniffer's Guide to Raw Traffic. [online]. [Accessed: 24/04/2022].
URL `http://yuba.stanford.edu/~casado/pcap/section1.html`