BRNO UNIVERSITY OF TECHNOLOGY
FACULTY OF INFORMATION TECHNOLOGY

IPK – 2nd Assignment

Packet Sniffer

MANUAL

18-04-2022

Václav Korvas (xkorva03)

Contents

Introduction

This manual describes usage, theory behind implementation and implementation details of packet sniffer.

Packet sniffer captures and filters packets on specific interface. According to the command line arguments provided to the program `UDP, ARP, TCP or ICMP` `n` number of packets are sniffed on specified port. The program will print destination and source MAC, IP address and ports to `stdout` with packet data in hexadecimal a ASCII format.

1 Build and usage

Whole program implementation is located in `ipk-sniffer.cpp` with all structure and function declarations and function descriptions located in `ipk-sniffer.h`. To be able to successfully build the project you need GNU Make. After you unpack the archive you can build the application using command `make` or `make debug` for additional info about captured packets. This will create executable file `ipk-sniffer` in project root directory. In attached Makefile are these targets: `all`, `debug` and `clean`, which removes all binary and executable files. Packet sniffer is possible to run with this command:

```
./ipk-sniffer [-i <interface>|--interface <interface>] {-p port}
{[--tcp|-t] [--udp|-u] [--arp] [--icmp]} {-n number} {-h|--help}
```

Individual parametr description:

- Parametr `-i` or `--interface` specifies on which interface are we going to listen. If given interface doesn't provide link-type other than `LINKTYPE_ETHERNET`[1] the program exits with return code 1.
- Parametr `-p` specifies on what port number on given interface will be packets filltered. If set port number can be in source or destination part. If not set all port numbers are considered. Port numbers can be in range 1-65535 if not the program exits with return code 1.
- Individual protocol parametrs specifies what protocols are we interested to sniff on. These parametrs are in **or** relation to each other. That means if we use for example parametrs `--tcp` and `--udp` we are interested in TCP or UDP packets. Parametr `--icmp` means we want to show ICMPv4 and ICMPv6 packets. If port parametr is set we implicitly assume protocols UDP and TCP.
- Parametr `-n` specified how many packets we want to sniff. Implicit value is 1.
- Parametr `-h` or `--help` prints simple help message and exits with return code 0.

All parametrs in curly brackets are optional and parametrs in square brackets are mandatory.

[1]LINKTYPE_ETHERNET has corresponding name `DLT_EN10MB`, which is used in code in function pcap_datalink.[1]

2 Packets

This next chapter deals with the individual packet protocols that we are interested in caputuring. It describes packet protocol headers and what informations we need and also on what OSI[2] layer these protocols operate.

First we need to talk about the ethernet frame [3], which operates on the second OSI layer (data link layer) more specific in type Ethernet II frame. Ethernet frame starts with ethernet header, which contains source and destination MAC addresses then there is data payload that includes headers for other protocols and ends with CRC checksum. We only accept ethernet type of data link layer protocol. And from the ethernet frame we will use destination and source addresses.

2.1 ARP

In this section we will talk about Address Resolution Protocol(ARP) [4] and informations that we will use from this packet protocol that is on third OSI layer (network layer)[3]. This protocol is used when a device that wants to determine the physical address of a device with a specific IP address (typically IPv4) sends an ARP request as a broadcast to its subnet. All devices on the subnet receive this ARP request, but only the device with the IP address specified in the ARP request will respond. Its response contains its own IP and MAC address.

From the header structure we will use `Sender hardware address (SHA)`,`Sender protocol address (SPA)`,`Target hardware address (THA)` and `Target protocol address (TPA)`.

2.2 IPv4 and IPv6

Internet Protocol version 4 (IPv4) [5] is the fourth version of the Internet Protocol IP. This protocol forms one of the basic communication frameworks of the Internet. IPV4 is a standardized version of the IP protocol that is used to create a communication framework on the Internet.

In the IPv4 header are these important informations that were used: `version` to make sure IPv4 packet was captured, `IHL` to calculate the position of the next header, because IPv4 header has variable size due to a optional field and `source IPv4 address` and `destination IPv4 address` and `protocol`, which defines what protocol is payload section. We are interested only in ICMP, UDP and TCP protocol numbers.

Internet Protocol version 6 (IPv6) [6] is a successor to IPv4. IPv6 has replaced its predecessor IPV4 primarily due to the exhaustion of IP address space. IPv6 introduces new address spaces. IPV6 address are 128-bits long and are typically written in hexadecimal in eight groups of four numbers separated by colons.

From the IPV6 header we use only the `source and destination IP addresses` and `next header`, which sets the upper layer protcol. From these protocols we are interested only in ICMPv6, UDP and TCP.

[2]Open Systems Interconnection model is a network model that has seven different layers [2]

[3]Older references placed ARP packet to data link layer, but the new ones places it into the network layer

2.2.1 ICMP

Internet Control Message Protocol(ICMP) [5] is on third OSI layer. ICMP is a supporting protocol in the Internet Protocol suite. Uses network devices including routers to send error messages and operational information. ICMP is part of the Internet protocol suite as defined in RFC 792. The ICMP packet is encapsulated in an IPv4 packet. The packet consists of header and data sections. The ICMP header starts after the IPv4 header and can be identified by IP protocol number 1.

From the datagram structure we will use only `ICMP type`. And from the IPv4 header we will use `source IP address` and `destination IP address`.

2.2.2 ICMPv6

Internet Control Message Protocol version 6 [6]. Is the ICMP implementation for for IPv6. ICMPv6 is a protocol operating on the network layer. ICMPv6 messages are transported by IPv6 packets in which the IPv6 Next Header and the value is number 58.

From the ICMPv6 packet we need only `ICMPv6 type` and from the IPv6 header we will need `source IP address` and `destination IP address`.

2.2.3 TCP

Transmission Control Protocol (TCP) [7] is a protocol operating on the fourth OSI layer (transport layer)[4]. TCP provides reliable and error-checked delivery.

The only thing needed from TCP segment header are `source port` and `destination port` numbers. This applies to both IPv4 and IPv6.

2.2.4 UDP

User Datagram Protocol (UDP) [8] is a protocol operating on the fourth OSI layer just like TCP. A user protocol for transmitting datagrams on networks, it is generally considered to be less reliable than, for example, TCP. UDP is suitable for deployments that require simplicity, for example in file sharing on LAN.

From UDP datagram header we will use `source port` and `destination port` numbers. This applies to both IPv4 and IPv6 as well.

[4]It is commonly known as TCP/IP because the original protocols of the suite are Transmitssion Control Protocol (TCP) and Internet Protocol (IP)

3 Implemtation details

The following section describes the structure of the program and the solution of individual problems. The program uses `pcap` [10] library and a few functions from this library, such as a function to create a filter for different packets or a function to read another packet. The beginning of how to capture packets and how to set up the packet capturer was inspired by this website [1].

3.1 Argument parsing

The first thing that is done after running the program is to parse the arguments and it's done in function `parse_arguments`. For this purpose, the function `get_optlong` is used from the `getopt.h` library.

If a user provides the `-i` or –interface argument without specifying a particular interface and also specifies other arguments but uses them incorrectly then the program will exit whit return code 1 without listing all the available interfaces. But if he uses other arguments and sets them correctly then the program will ignore the other arguments and prints all interfaces.

Handling the interfaces argument was a little tricky. So instead of making it an optional argument I made it a mandatory argument and when the interfaces was not specified I handled the return value from `get_optlong`.

3.2 Printing all interfaces

If all input arguments are set correctly and if the `-i` or `--interface` argument was not used or was used but no specific interface was specified, all available interfaces are listed.

To find all interfaces I used function `pcap_findalldevs` from the pcap library [10]. This fills structure `pcap_if_t` which is a linked-list. So then I just loop through it and print each name on individual line. The the function exits with return code 0. This all happens in function `print_interface_devices`.

3.3 Preparations before caputuring

First thing that is done is using function `pcap_open_live`[10] to open specified device for caputuring. To this function we specifie that we want to capture packet in promiscuos mode and set the buffer timeout to 100ms. Then we make sure that our device support ethernet link-layer headers. If not then the program exits with error code 1.

After that we need to create filter expression based on the program arguments. We need to correctly combine input protocol types to our string expression. This is done in function `create_filter`. Then we just compiles our filter expression and we are ready to capture packets.

3.4 Packet processing

Individual packet processing and "unwrapping" single packet layers happens in several functions. The main function `process_packet` is called from main in `pcap_loop` [10] for n-number of packets.

The first thing that happens is that the source and destination MAC addresses are stored in structure `output_data_t` created for this purpose. Then it's determined what type of protocol is in ethernet payload if it's IPv4, IPv6 or ARP, according to this corresponding functions are called. After the protocol is determined we move 14 bytes[5] to get to the next protocol header.

IPv4 and IPv6 are handled in separated functions. In these functions are stored informations about source and destination IP addresses and port numbers. Also in these functions is determined if IPv4 and IPv6 encapsulates UDP, TCP or ICMP packets.

Then as we move to above layers more informations are stored in the `output_data_t` structure and at the end of the functions these informations are printed alongside the whole packet. ICMP, ICMPv6 and ARP packets are printed withou port numbers. Otherwise all printed types of informations are the same.

3.5 Output

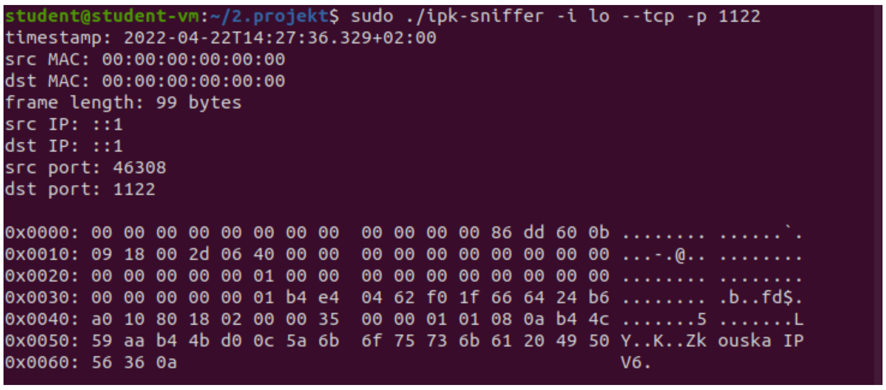
The output of the program is printed to stdout and tries to be similar to the output of Wireshark. First the timestamp[6] is printed. then the source and destination MAC address, then the frame length in bytes, the source and destination IP address and if it is TCP or UDP protocols, the source and destination port. Each of this information is printed on its own line. Finally, the total contents of the packet are listed in hexadecimal and ascii format. This output is the same as in wireshark with a larger space after 8 bytes.

4 Testing

Testing was done manually. Each time a packet was captured and output, I manually compared this output with the output of Wireshark [13]. In order not to accidentally confuse the packet, I used filtering in Wireshark using individual protocols. The program was tested on both the reference machine and the local machine both unix based systems. The output always matched the Wireshark output. The program was also tested several times on leaking to see if all sources were correctly released and no errors were found here either. The ICMP and ICMPv6 were tested using `ping` and `ping -6`. And IPv6 packets were tested using the utility `nc`, where I opened TCP connection on some port and captured the packets on loopback interface.

[5]Ethernet header has static length and it's 14 bytes

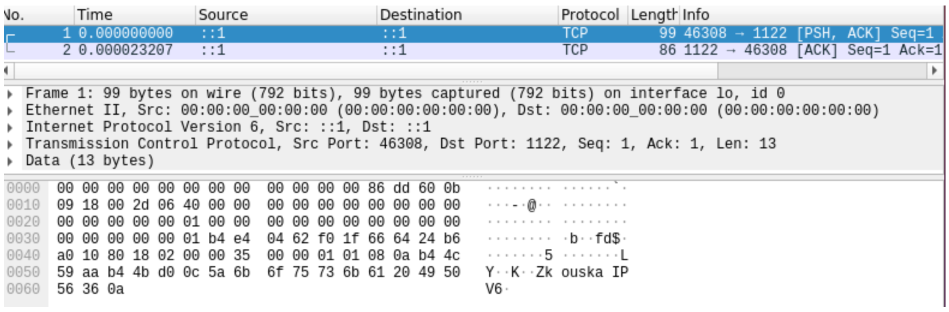
[6]How to print the timestamp was a bit problem, eventually I came across a solution on stackoverflow and codes from Sebastian and K. Haskins where taken and combined together. Original codes can be found here [11, 12]



```
student@student-vm:~/2.projekt$ sudo ./ipk-sniffer -i lo --tcp -p 1122
timestamp: 2022-04-22T14:27:36.329+02:00
src MAC: 00:00:00:00:00:00
dst MAC: 00:00:00:00:00:00
frame length: 99 bytes
src IP: ::1
dst IP: ::1
src port: 46308
dst port: 1122

0x0000: 00 00 00 00 00 00 00 00  00 00 00 00 86 dd 60 0b ........ ......`.
0x0010: 09 18 00 2d 06 40 00 00  00 00 00 00 00 00 00 00 ...-.@.. ........
0x0020: 00 00 00 00 00 01 00 00  00 00 00 00 00 00 00 00 ........ ........
0x0030: 00 00 00 00 00 01 b4 e4  04 62 f0 1f 66 64 24 b6 ........ .b..fd$.
0x0040: a0 10 80 18 02 00 00 35  00 00 01 01 08 0a b4 4c .......5 .......L
0x0050: 59 aa b4 4b d0 0c 5a 6b  6f 75 73 6b 61 20 49 50 Y..K..Zk ouska IP
0x0060: 56 36 0a                                         V6.
```

Figure 1: Packet captured by ipk-sniffer. Packet with IPv6 protocol on specified port.

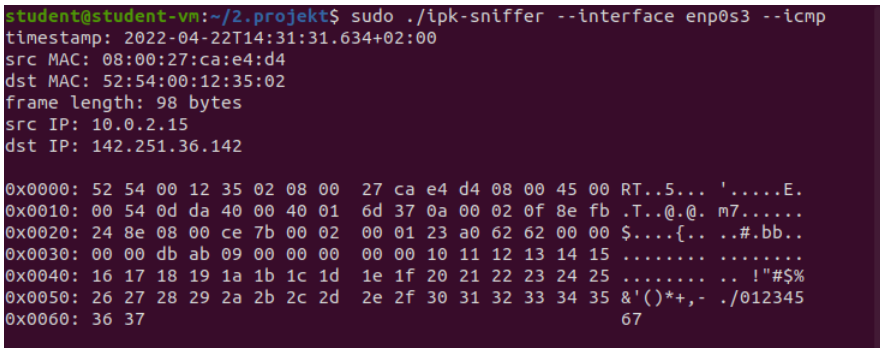


```
No.     Time          Source    Destination   Protocol Length Info
      1 0.000000000   ::1       ::1           TCP        99 46308 → 1122 [PSH, ACK] Seq=1
      2 0.000023207   ::1       ::1           TCP        86 1122 → 46308 [ACK] Seq=1 Ack=1

Frame 1: 99 bytes on wire (792 bits), 99 bytes captured (792 bits) on interface lo, id 0
Ethernet II, Src: 00:00:00_00:00:00 (00:00:00:00:00:00), Dst: 00:00:00_00:00:00 (00:00:00:00:00:00)
Internet Protocol Version 6, Src: ::1, Dst: ::1
Transmission Control Protocol, Src Port: 46308, Dst Port: 1122, Seq: 1, Ack: 1, Len: 13
Data (13 bytes)

0000  00 00 00 00 00 00 00 00  00 00 00 00 86 dd 60 0b   ········ ······`·
0010  09 18 00 2d 06 40 00 00  00 00 00 00 00 00 00 00   ···-·@·· ········
0020  00 00 00 00 00 01 00 00  00 00 00 00 00 00 00 00   ········ ········
0030  00 00 00 00 00 01 b4 e4  04 62 f0 1f 66 64 24 b6   ········ ·b··fd$·
0040  a0 10 80 18 02 00 00 35  00 00 01 01 08 0a b4 4c   ·······5 ·······L
0050  59 aa b4 4b d0 0c 5a 6b  6f 75 73 6b 61 20 49 50   Y··K··Zk ouska IP
0060  56 36 0a                                           V6·
```

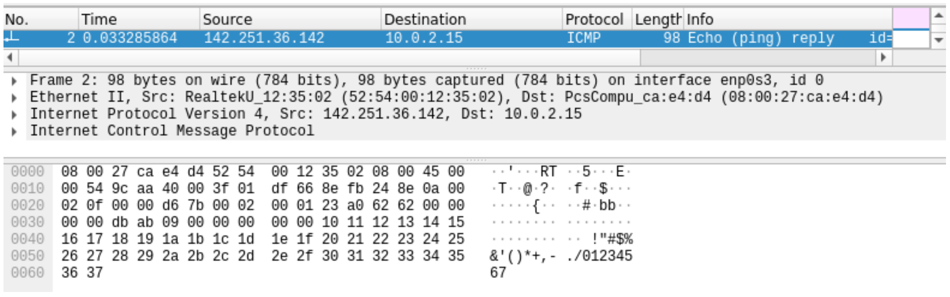
Figure 2: Same IPv6 packet captured with Wireshark



```
student@student-vm:~/2.projekt$ sudo ./ipk-sniffer --interface enp0s3 --icmp
timestamp: 2022-04-22T14:31:31.634+02:00
src MAC: 08:00:27:ca:e4:d4
dst MAC: 52:54:00:12:35:02
frame length: 98 bytes
src IP: 10.0.2.15
dst IP: 142.251.36.142

0x0000: 52 54 00 12 35 02 08 00  27 ca e4 d4 08 00 45 00 RT..5... '.....E.
0x0010: 00 54 0d da 40 00 40 01  6d 37 0a 00 02 0f 8e fb .T..@.@. m7......
0x0020: 24 8e 08 00 ce 7b 00 02  00 01 23 a0 62 62 00 00 $....{.. ..#.bb..
0x0030: 00 00 db ab 09 00 00 00  00 00 10 11 12 13 14 15 ........ ........
0x0040: 16 17 18 19 1a 1b 1c 1d  1e 1f 20 21 22 23 24 25 ........ .. !"#$%
0x0050: 26 27 28 29 2a 2b 2c 2d  2e 2f 30 31 32 33 34 35 &'()*+,- ./012345
0x0060: 36 37                                            67
```

Figure 3: Packet captured by ipk-sniffer. Packet with ICMP protocol.



```
No.   Time         Source          Destination   Protocol Length Info
    2 0.033285864  142.251.36.142  10.0.2.15     ICMP         98 Echo (ping) reply    id=
```

```
Frame 2: 98 bytes on wire (784 bits), 98 bytes captured (784 bits) on interface enp0s3, id 0
Ethernet II, Src: RealtekU_12:35:02 (52:54:00:12:35:02), Dst: PcsCompu_ca:e4:d4 (08:00:27:ca:e4:d4)
Internet Protocol Version 4, Src: 142.251.36.142, Dst: 10.0.2.15
Internet Control Message Protocol
```

```
0000  08 00 27 ca e4 d4 52 54  00 12 35 02 08 00 45 00   ··'···RT ··5···E·
0010  00 54 9c aa 40 00 3f 01  df 66 8e fb 24 8e 0a 00   ·T··@·?· ·f··$···
0020  02 0f 00 00 d6 7b 00 02  00 01 23 a0 62 62 00 00   ·····{·· ··#·bb··
0030  00 00 db ab 09 00 00 00  00 00 10 11 12 13 14 15   ········ ········
0040  16 17 18 19 1a 1b 1c 1d  1e 1f 20 21 22 23 24 25   ········ ·· !"#$%
0050  26 27 28 29 2a 2b 2c 2d  2e 2f 30 31 32 33 34 35   &'()*+,- ./012345
0060  36 37                                              67
```

Figure 4: Same ICMP packet captured with Wireshark

References

[1] Carstens, T., Guy, H.. *Programming with pcap* [online]. 2022. [accessed 2022-04-17]. Available from: `https://www.tcpdump.org/pcap.html`

[2] Wikipedia contributors. *OSI model* [online]. Wikipedia, The Free Encyclopedia; 2022-04-14. [accessed 2022-04-17]. Available from: `https://en.wikipedia.org/wiki/OSI_model`

[3] Wikipedia contributors. *Ethernet frame* [online]. Wikipedia, The Free Encyclopedia; 2022-04-01. [accessed 2022-04-17]. Available from: `https://en.wikipedia.org/wiki/Ethernet_frame`

[4] Wikipedia contributors. *Address Resolution Protocol* [online]. Wikipedia, The Free Encyclopedia; 2022-04-06. [accessed 2022-04-17]. Available from: `https://en.wikipedia.org/wiki/Address_Resolution_Protocol`

[5] Veselý, V. *Síťová vrstva – IPv4* [University lecture]. 2021. Available from: `https://wis.fit.vutbr.cz/FIT/st/cfs.php.cs?file=%2Fcourse%2FIPK-IT%2Flectures%2FIPK2021-06-IPv6.pdf&cid=14678`

[6] Veselý, V.. *IPv6 síťová vrstva* [University lecture]. 2021. Available from: `https://wis.fit.vutbr.cz/FIT/st/cfs.php.cs?file=%2Fcourse%2FIPK-IT%2Flectures%2FIPK2021-04-IPv4.pdf&cid=14678`

[7] Wikipedia contributors. *Transmission Control Protocol* [online]. Wikipedia, The Free Encyclopedia; 2022-04-09. [accessed 2022-04-17]. Available from: `https://en.wikipedia.org/wiki/Transmission_Control_Protocol`

[8] Wikipedia contributors. *User Datagram Protocol* [online]. Wikipedia, The Free Encyclopedia; 2022-03-24. [accessed 2022-04-17]. Available from: `https://en.wikipedia.org/wiki/User_Datagram_Protocol`

[9] Free Software Foundation. *getopt(3) — Linux manual page* [online], [accessed 2022-04-17]. Available from: `https://www.man7.org/linux/man-pages/man3/getopt.3.html`

[10] Jacobson, V., et al. *Man page of pcap* [online], 2020-09-09 [accessed 2022-04-17]. Available from: `https://www.tcpdump.org/manpages/pcap.3pcap.html`

[11] Sebastian. *C++ RFC3339 timestamp with milliseconds using std::chrono* [online], [accessed 2022-04-17]. Available from: `https://stackoverflow.com/questions/54325137/c-rfc3339-timestamp-with-milliseconds-using-stdchrono`

[12] Haskins, K. *I'm trying to build an RFC3339 timestamp in C* [online],[accessed 2022-04-17]. Available from: `https://stackoverflow.com/questions/48771851/im-trying-to-build-an-rfc3339-timestamp-in-c-how-do-i-get-the-timezone-offset`

[13] Combs, G. *Wireshark* [online] 2022,[accessed 2022-04-17]. Available from: `https://www.wireshark.org/`