

Brno University of Technology
Faculty of Information Technology



Computer Communications and Networks
2021/2022

Project documentation
Variant ZETA: Packet Sniffer

Table of contents

Table of contents	1
Introduction	2
Task description	2
Technical requirements	2
Implementation	3
How the application works	3
Compile	3
Program start	3
pcap_loop()	3
Callback function got_packet()	4
Variable declaration	4
Timestamp constructing and displaying data	5
Pic: example of HexDump data	5
Sample outputs	6
Testing	8
Summary	8
netcat testing	9
Test case #1: tcp packet with port filter using IPv6	9
Test case #2: udp packet with port filter using IPv6	10
ping testing	12
Test case #3: ICMPv6 packet receiving	12
Test case #4: ICMPv4 packet receiving	13
List of sources	15

Introduction

Task description

The problem of the project was to develop an application - a network analyzer that can receive and filter packets (depending on the protocol: TCP, UDP, ICMP, ARP) on a particular device. Output data are written to the standard output (stdout) in Hex Dump format (analog is pcapng file, which is used in Wireshark). Before that, the display shows the basic values for the current packet: Packet timestamp, source and destination MAC addresses, frame length, source and destination IP addresses, source and destination ports.

Technical requirements

The program should be called as follows:

```
$ sudo ./ipk-sniffer {-h} [-i interface | --interface interface] {-p port}
{[--tcp|-t] [--udp|-u] [--arp] [--icmp] } {-n num}
```

Where:

- **-h**: will write a help message.
- **-i or --interface interface**: just one interface on which to listen. If this parameter is not specified, or if only **-i** is specified without a value, a list of active interfaces will be printed.
- **-p port**: will filter packets on a given interface by port; if this parameter is not specified, all ports are considered; if the parameter is specified, the port can occur in both the source and destination part.
- **-t or --tcp**: will display only TCP packets.
- **-u or --udp**: will display only UDP packets.
- **--icmp**: will display only ICMPv4 and ICMPv6 packets.
- **--arp**: will display only ARP frames.
- **-n num**: specifies the number of packets to be displayed; if not specified, consider displaying only one packet (as if **-n 1**).

Also:

- Unless specific protocols are specified (or if all of them are listed at once), all protocols (i.e. all content, regardless of the protocol) are considered for printing.
- The program can be correctly terminated at any time using Ctrl+C.

Implementation

How the application works

The core of the application is the library `pcap`. It is a portable platform for low-level network monitoring that uses the standard pcap format. The libpcap interface supports a filtering mechanism called BPF [\[1\]](#), which was used in the current project. It provides a raw interface to data link layers, permitting raw link-layer packets to be sent and received. Also to work with certain protocols (work with data structures) we used headers from the `netinet` library.

Compile

In the root folder along with the source code there is a Makefile, which allows you to compile the program simply from command line:

```
$ make
```

Output executable file is `./ipk-sniffer` (***require root to run properly!***). Recommended usage:

```
$ sudo ./ipk-sniffer ...
```

Program start

The main function of the program processes the arguments, "assembles" the filter based on the prepared arguments and starts the sniffer function.

First, the `parameters_parsing()` function is called, divided into two parts: processing of short arguments ("`-`") and processing of long arguments ("`--`"). After that, all arguments are checked in the same function to make sure they fit the conditions of the application.

Next, the `filter_ctor()` function is called, which translates the user's arguments into the appropriate `pcap_filter` format.

The `pcap_t *handle` variable is the descriptor of the packet capture endpoint. Then it is passed for processing to the `handling_pcap()` function, which consists of several parts:

- The variable `device` (which stores the values of the user parameter `interface`) is checked and, if it is empty, a list of all available devices is displayed.
- If the user has set the interface value, the program tries to open it with `pcap_open_live()`
- `pcap_lookupnet()` returns the subnet mask and the address of the network where packets will be listened to. The mask is further used to set the filter.
- `pcap_compile()` and `pcap_setfilter()` compile and install the program filter.

pcap_loop()

If the previous processing went without errors, it means that all the input data were prepared to start listening to the packets. `pcap_loop()` is one of three existing and one of two automatically working functions to get packets. It takes the `handle` descriptor, the number of packets we want to handle and

the packet capture call back function as parameters. [2] For each packet received, `pcap_loop()` calls the handler – in my implementation this is the `got_packet()` function.

Callback function `got_packet()`

This function is one of the most important parts of the program, along with `handling_pcap()`, where we "built" the network adapter, and `pcap_loop()`, which automatically reads packets from the adapter. It is used to collect and construct all the necessary data about the current packet, such as:

- Current timestamp
- MAC addresses for source and destination
- Length of current data frame
- IP addresses for source and destination
- Source and destination ports
- The data of the package itself

Variable declaration

For our work we need data structures from the `netinet` library:

- `<netinet/udp.h>`
- `<netinet/tcp.h>`
- `<netinet/ip.h>`
- `<netinet/ip6.h>`
- `<netinet/if_ether.h>`

`struct ip* iphdr` and `struct ip6_hdr* ip6hdr` are used to process IP packets. `struct tcphdr* tcphdr` and `struct udphdr* udphdr` are used to collect packet ports depending on the selected protocol.

Depending on the packet type (ARP, IPv4/6) we get the current protocol used in the packet (only in the case of IPv4/6) and the source and destination IP addresses. We can obtain current packet type by declaring a structure:

```
struct ether_header *etherhdr = (struct ether_header *)packet;
uint16_t ether_type = ntohs(etherhdr->ether_type);
```

Therefore `ether_type` could have 3 values (which can be recognized by application): `ETHERTYPE_ARP`, `ETHERTYPE_IP`, `ETHERTYPE_IPV6`. Also we can retrieve source and destination MAC addresses by sprinting 6 values from `etherhdr->ether_shost[6]` and `etherhdr->ether_dhost[6]` each. Then, having a protocol, we are able to proceed to select the protocol according to what we obtained above. I've already defined required values, where `PROTOCOL_TCP = 6` and `PROTOCOL_UDP = 17`. [3] Use `switch(protocol)` to select the actual protocol a packet is using. Now we can get UDP and TCP ports:
`sourcePort = ntohs(tcphdr->th_sport) OR ntohs(udphdr->uh_sport)`

And the same for `destinationPort`.

Timestamp constructing and displaying data

At the beginning of the function, a `char *timestamp` variable is created, which, with the argument `struct pcap_pkthdr header->ts` gets its value from the `timestamp_ctor()` function. The `timeval` structure we get as an argument carries the value `tv_sec`, which we can translate into milliseconds. [4] Returned timestamp is in *RFC3339* format.

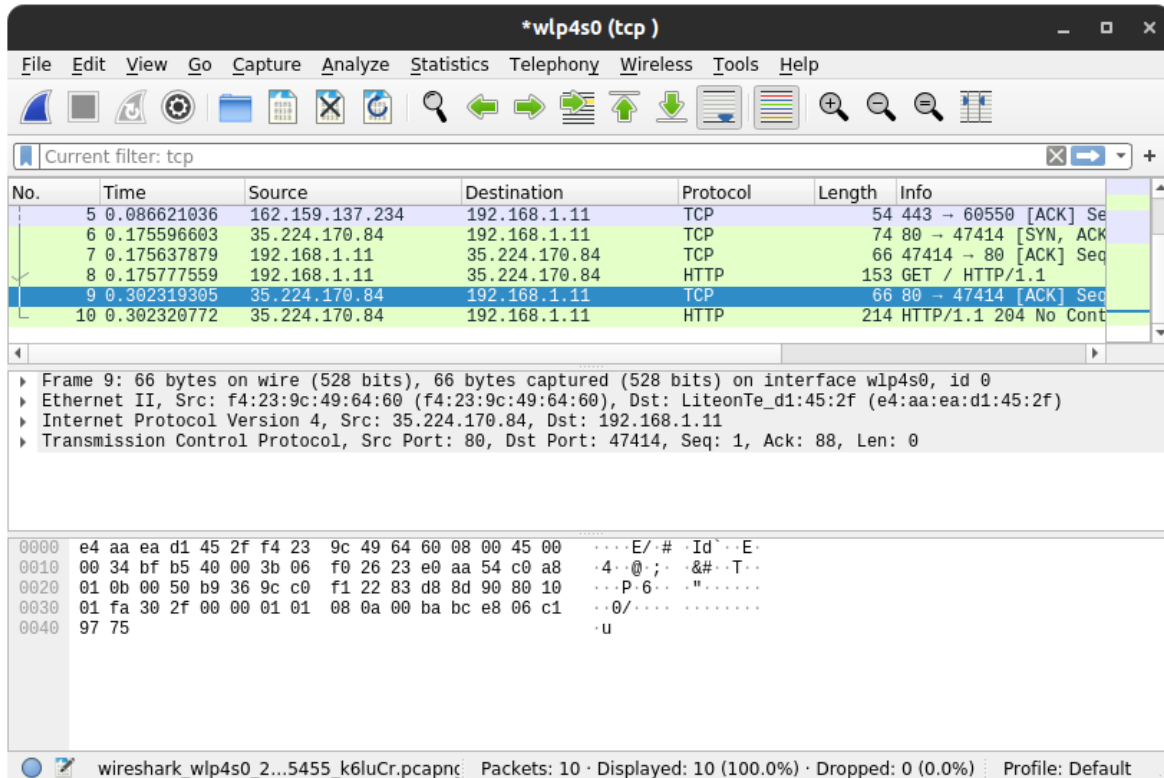
After processing the packet completely (if it passed without errors), we call the `display_packet_data()` function and pass in it all the necessary parameters that are required for basic information about the packet. Then `display_packet_dump()` is called to display the Hex Dump of a packet, using `const u_char* packet` and `const int len` as parameters.

```
0000 e4 aa ea d1 45 2f f4 23 9c 49 64 60 08 00 45 00  ...E/# ·Id`·E·
0010 00 a6 88 b7 40 00 37 11 c2 c0 42 16 f4 05 c0 a8  ...@·7· ·B····
0020 01 0b c3 5e 95 3d 00 92 6a 5d 90 78 0d ee ab 03  ...^·=· ·j]·x···
0030 c1 42 00 02 f4 4d be de 00 01 92 7f df cb 9a b9  ·B··M········
0040 fa 7a 6e 62 5a 52 ac 20 04 b4 c2 79 32 c5 93 b4  ·znbZR· ··y2··
0050 34 c7 b3 38 f0 93 bf 06 4a 3c e0 e1 be 7b 33 f8  4·8···J<··{3·
0060 42 bb e9 d9 74 3c 35 d8 7d f5 01 b6 5c f0 04 9d  B··t<5· }··\··
0070 5a d2 36 c1 f4 a8 f9 0a cb b6 dd b3 3e 18 26 21  Z·6····>·&!·
0080 cf 8a f8 59 26 1c 66 73 60 20 32 e6 54 65 5b 8e  ·Y&·fs ` 2·Te[·
0090 c4 8e b5 d0 81 f8 f6 48 64 10 42 5c 41 03 3a f7  ·····H d·B\A·:·
00a0 4e 31 65 68 7e 7d 5d d4 34 87 90 70 a2 d5 88 9c  N1eh~}]· 4·p···
00b0 55 60 00 00  U`··
```

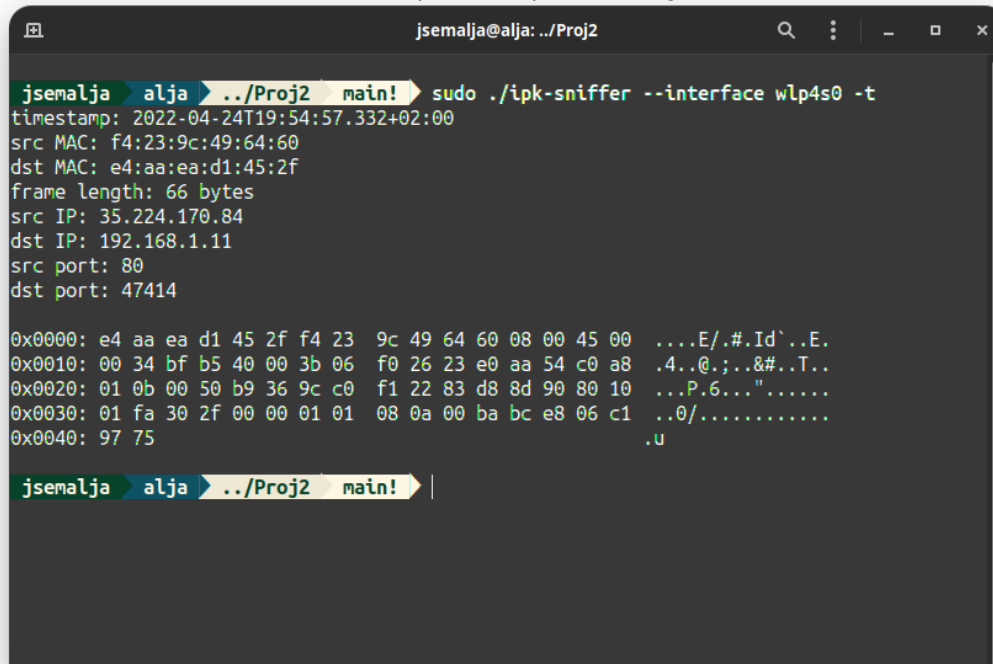
Pic: example of HexDump data

Sample outputs

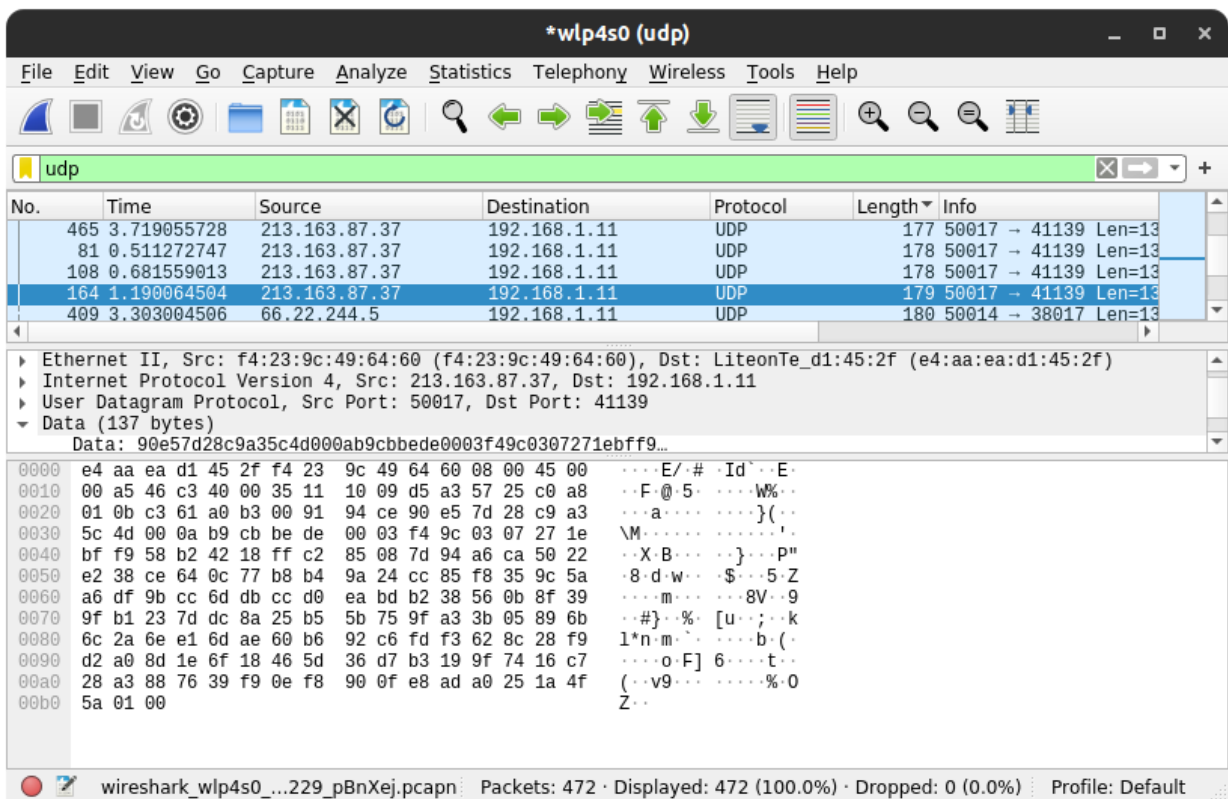
Below there are simple outputs examples using Wireshark to compare received packets.



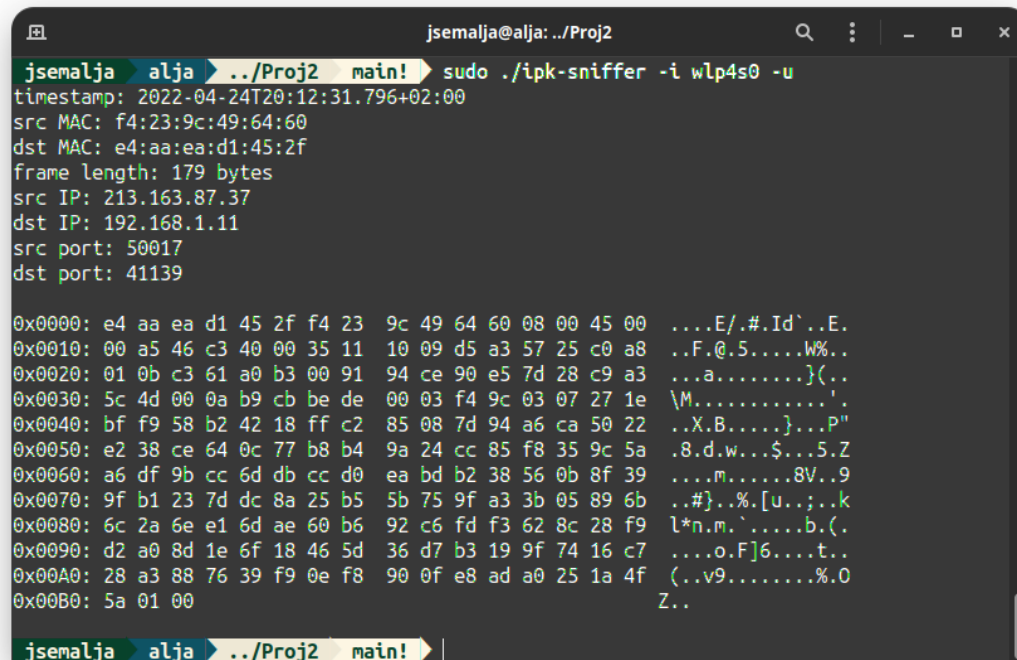
Pic: one TCP packet captured using Wireshark



Pic: the same TCP packet captured using application



Pic: one UDP packet captured using Wireshark



Pic: the same UDP packet captured using application

Testing

For tests I used `nc`, `ping`, `ping6`.

Summary

To test using `netcat` I needed two command line sessions: first one is for listening and second one is for sending some data packets. [\[5\]](#) In tests cases #1-4 for examination `netcat` was used with parameters such as:

```
$ nc -6 [-l | localhost] 8080
```

Where command with the argument `-l` is the “server” and the other one with `localhost` is a “client”. The task of the application will be to “intercept” packets between this server and the client. The same rules apply to packets with the UDP protocol.

```
$ nc -u -6 [-l | localhost] 8080
```

Second function I used for testing is `ping` (and `ping6`). `ping` sending an ICMP ECHO_REQUEST to network hosts. [\[6\]](#)

netcat testing

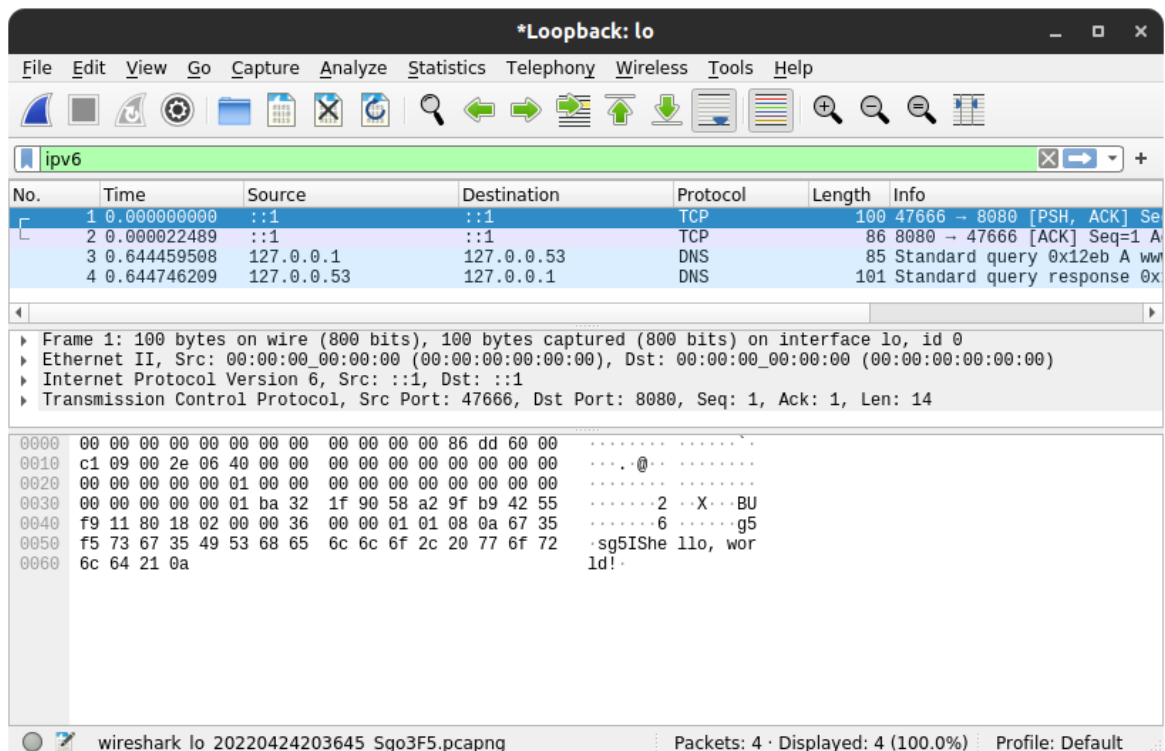
Test case #1: tcp packet with port filter using IPv6

Connection is OK at port 8080.

Wireshark also can see this packet.

```
jsemalja@alja: ~  
jsemalja alja ➤ nc -6 -l 8080  
hello, world!
```

```
jsemalja alja ➤ nc -6 localhost 8080  
hello, world!
```



```
jsemalja@alja: ~/Proj2  
jsemalja alja ➤ ../Proj2 main! ➤ sudo ./ipk-sniffer -i lo -t -p 8080  
timestamp: 2022-04-24T20:36:48.276+02:00  
src MAC: 00:00:00:00:00:00  
dst MAC: 00:00:00:00:00:00  
frame length: 100 bytes  
src IP: ::1  
dst IP: ::1  
src port: 47666  
dst port: 8080  
  
0x0000: 00 00 00 00 00 00 00 00 00 00 00 00 86 dd 60 00 .....`.  
0x0010: c1 09 00 2e 06 40 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 .....  
0x0030: 00 00 00 00 00 01 ba 32 1f 90 58 a2 9f b9 42 55 .....2..X...BU  
0x0040: f9 11 80 18 02 00 00 36 00 00 01 01 08 0a 67 35 .....6.....g5  
0x0050: f5 73 67 35 49 53 68 65 6c 6c 6f 2c 20 77 6f 72 ..sg5IShe llo, wor  
0x0060: 6c 64 21 0a .....ld!..  
  
jsemalja alja ➤ ../Proj2 main!
```

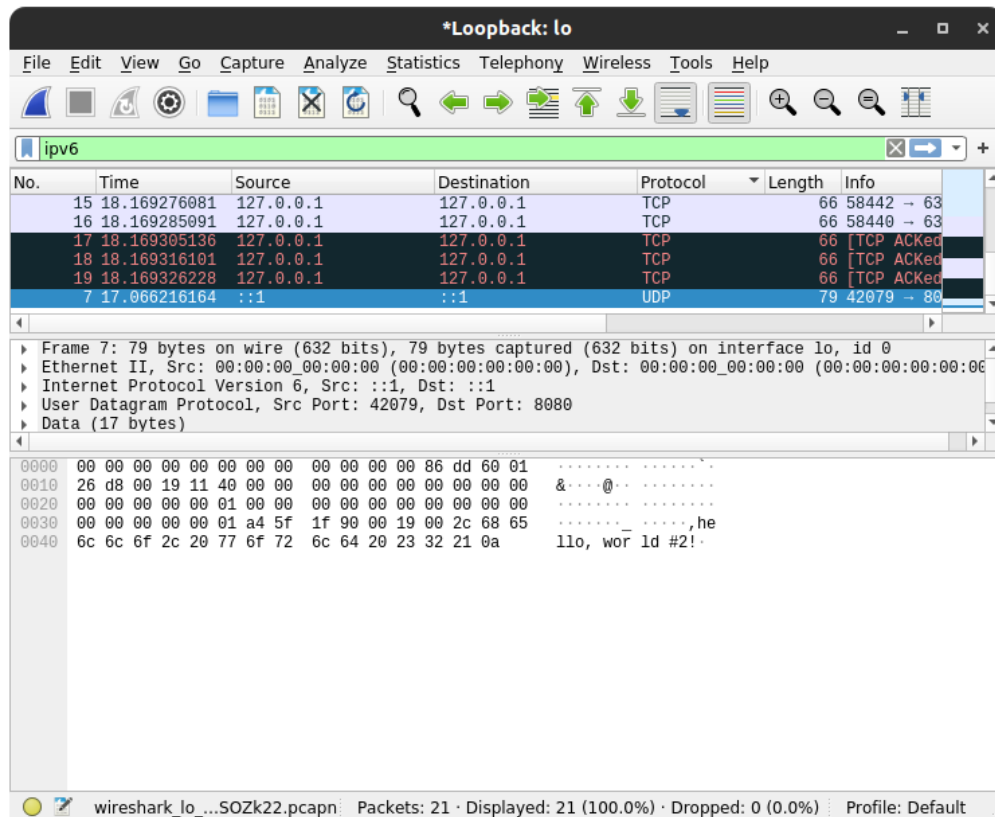
Application as well shows the received packet.

TEST CASE #1 IS OK

Test case #2: udp packet with port filter using IPv6

```
jsemalja@alja: ~  
jsemalja alja ~ nc -u -6 -l 8080  
hello, world #2!
```

```
jsemalja@alja: ~  
jsemalja alja ~ nc -u -6 localhost 8080  
hello, world #2!
```



Connection is OK, packet transmission is OK.

```
jsemalja@alja: ../Proj2 main! sudo ./ipk-sniffer -i lo -u -p 8080
timestamp: 2022-04-24T20:52:03.444+02:00
src MAC: 00:00:00:00:00:00
dst MAC: 00:00:00:00:00:00
frame length: 79 bytes
src IP: ::1
dst IP: ::1
src port: 42079
dst port: 8080

0x0000: 00 00 00 00 00 00 00 00 00 00 00 00 86 dd 60 01 .....`.
0x0010: 26 d8 00 19 11 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 a4 5f 1f 90 00 19 00 2c 68 65 ....._.....,he
0x0040: 6c 6c 6f 2c 20 77 6f 72 6c 64 20 23 32 21 0a llo, world #2!.

jsemalja@alja: ../Proj2 main! |
```

Application received a packet.

TEST CASE #2 IS OK

ping testing

Test case #3: ICMPv6 packet receiving

```
jsemalja@alja: ~  
jsemalja alja ~$ ping6 -c 1 ::1  
PING ::1(::1) 56 data bytes  
64 bytes from ::1: icmp_seq=1 ttl=64 time=0.051 ms  
  
--- ::1 ping statistics ---  
1 packets transmitted, 1 received, 0% packet loss, time 0ms  
rtt min/avg/max/mdev = 0.051/0.051/0.051/0.000 ms
```

*Loopback: lo

File Edit View Go Capture Analyze Statistics Telephony Wireless Tools Help

ipv6

No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000000	::1	::1	ICMPv6	118	Echo (ping) request
2	0.000016762	::1	::1	ICMPv6	118	Echo (ping) reply
3	6.588112378	127.0.0.1	127.0.0.53	DNS	87	Standard query 0
4	6.588747802	127.0.0.1	127.0.0.53	DNS	81	Standard query 0
5	6.602170270	127.0.0.53	127.0.0.1	DNS	103	Standard query response
6	6.602304228	127.0.0.53	127.0.0.1	DNS	97	Standard query response

Frame 1: 118 bytes on wire (944 bits), 118 bytes captured (944 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
Internet Control Message Protocol v6

```
0000 00 00 00 00 00 00 00 00 00 00 00 86 dd 60 07 .....`.  
0010 84 f7 00 40 3a 40 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 .....  
0030 00 00 00 00 00 01 80 00 41 c1 00 16 00 01 00 9e .....A.....  
0040 65 62 00 00 00 00 0c d8 0c 00 00 00 00 00 10 11 eb.....  
0050 12 13 14 15 16 17 18 19 1a 1b 1c 1d 1e 1f 20 21 .....!  
0060 22 23 24 25 26 27 28 29 2a 2b 2c 2d 2e 2f 30 31 "#$%&'()*+,-./01  
0070 32 33 34 35 36 37 234567
```

wireshark_lo_...6Dt2WH.pcapng Packets: 6 · Displayed: 6 (100.0%) · Dropped: 0 (0.0%) Profile: Default

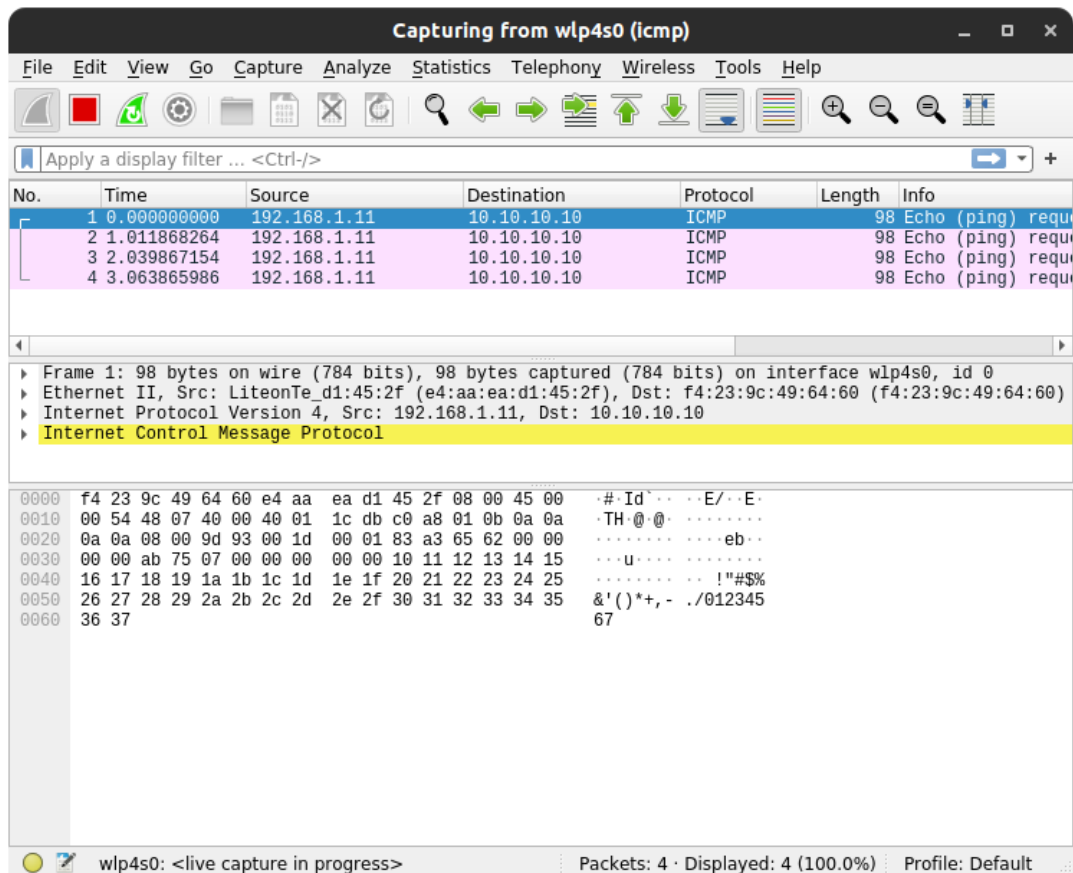
```
jsemalja@alja: ../Proj2  
jsemalja alja ../Proj2 main!$ sudo ./ipk-sniffer -i lo --icmp  
timestamp: 2022-04-24T20:59:12.884+02:00  
src MAC: 00:00:00:00:00:00  
dst MAC: 00:00:00:00:00:00  
frame length: 118 bytes  
src IP: ::1  
dst IP: ::1  
src port: 0  
dst port: 0  
  
0x0000: 00 00 00 00 00 00 00 00 00 00 00 00 86 dd 60 07 .....`.  
0x0010: 84 f7 00 40 3a 40 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 .....  
0x0030: 00 00 00 00 00 01 80 00 41 c1 00 16 00 01 00 9e .....A.....  
0x0040: 65 62 00 00 00 00 0c d8 0c 00 00 00 00 00 10 11 eb.....  
0x0050: 12 13 14 15 16 17 18 19 1a 1b 1c 1d 1e 1f 20 21 .....!  
0x0060: 22 23 24 25 26 27 28 29 2a 2b 2c 2d 2e 2f 30 31 "#$%&'()*+,-./01  
0x0070: 32 33 34 35 36 37 234567
```

As well as wireshark, application also can see and receive packet and its data.

TEST CASE #3 IS OK

Test case #4: ICMPv4 packet receiving

```
jsemalja@alja: ~  
jsemalja alja ~ ping 10.10.10.10  
PING 10.10.10.10 (10.10.10.10) 56(84) bytes of data.  
^C  
--- 10.10.10.10 ping statistics ---  
4 packets transmitted, 0 received, 100% packet loss, time 30  
64ms
```



As we can see in Wireshark, data from ping was successfully transmitted.

```
jsemalja@alja: ../Proj2 main! sudo ./ipk-sniffer -i wlp4s0 --icmp
timestamp: 2022-04-24T21:22:44.440+02:00
src MAC: e4:aa:ea:d1:45:2f
dst MAC: f4:23:9c:49:64:60
frame length: 98 bytes
src IP: 192.168.1.11
dst IP: 10.10.10.10
src port: 0
dst port: 0

0x0000: f4 23 9c 49 64 60 e4 aa ea d1 45 2f 08 00 45 00 .#.Id`....E/..E.
0x0010: 00 54 48 07 40 00 40 01 1c db c0 a8 01 0b 0a 0a .TH.@.@.....
0x0020: 0a 0a 08 00 9d 93 00 1d 00 01 83 a3 65 62 00 00 .....eb..
0x0030: 00 00 ab 75 07 00 00 00 00 00 10 11 12 13 14 15 ...u.....
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

jsemalja@alja: ../Proj2 main! |
```

The same packet received by application.

TEST CASE #4 IS OK

List of sources

1. Berkeley Packet Filter - *Wikipedia* [online]. Available from: [Berkeley Packet Filter - Wikipedia](#)
2. *Home | TCPDUMP & LIBPCAP* [online]. Copyright © 2010 [cit. 24.04.2022]. Available from: [pcap_loop\(3PCAP\) man page | TCPDUMP & LIBPCAP](#)
3. *Internet Assigned Numbers Authority* [online]. Available from: [Protocol Numbers](#)
4. `<sys/time.h>`. *The Open Group Publications Catalog* [online]. Copyright © 1997 The Open Group [cit. 24.04.2022]. Available from: [<sys/time.h>](#)
5. *Ubidots Help Center* [online]. Available from: [Sending TCP/UDP packets using Netcat | Ubidots Help Center](#)
6. ping6(8) - Linux man page. *Linux Documentation* [online]. Available from: [ping6\(8\) - Linux man page](#)
7. EtherType - *Wikipedia* [online]. Available from: [EtherType - Wikipedia](#)
8. Develop a Packet Sniffer with Libpcap - vichargrave.github.io. *vichargrave.github.io* [online]. Copyright © 2022 Vic Hargrave. Powered by [cit. 24.04.2022]. Available from: [Develop a Packet Sniffer with Libpcap](#)
9. Programming with pcap | TCPDUMP & LIBPCAP. *Home | TCPDUMP & LIBPCAP* [online]. Copyright © 2010 [cit. 24.04.2022]. Available from: [Programming with pcap | TCPDUMP & LIBPCAP](#)
10. C - pcap_findalldevs to display all interfaces is stuck in an infinite loop - Stack Overflow. *Stack Overflow - Where Developers Learn, Share, & Build Careers* [online]. Available from: [C - pcap_findalldevs to display all interfaces is stuck in an infinite loop - Stack Overflow](#)
11. c - how to get hexdump of a structure data - Stack Overflow. *Stack Overflow - Where Developers Learn, Share, & Build Careers* [online]. Available from: [how to get hexdump of a structure data - Stack Overflow](#)
12. time - I'm trying to build an RFC3339 timestamp in C. How do I get the timezone offset? - Stack Overflow. *Stack Overflow - Where Developers Learn, Share, & Build Careers* [online]. Available from: [I'm trying to build an RFC3339 timestamp in C. How do I get the timezone offset? - Stack Overflow](#)