Brno University of Technology

Faculty of Information Technology

Computer Communication and Networks

Project no. 2 documentation

ZETA: Packet Sniffer

Jan Zbořil

xzbori20@stud.fit.vutbr.cz

April 2021

Table of Contents

1. Program ipk-sniffer

Goal of this project has been to create a working console user interface program, which is able to sniff network traffic on one of the devices interface and display captured information in form of ASCII and HEX output.

Program was created using C++ and C programming languages in VS Code IDE. Compilation is done via Makefile located in root directory.

1.1 Running the program

Program C++ code has to be compiled using given Makefile using command make. Compiled program can be run using command line. Command to run program is ./ipk-sniffer. Due to nature of this project it is advised to run this program as an administrator.

Program accepts these arguments:

- -i/--interface
  - specifies on which interface the traffic is to be captured
  - accepts one string parameter
- -p
  - specifies if the traffic is to be captured only on one selected port
  - accepts one string parameter, specifying more ports is forbidden
- -t/--tcp
  - capture only TCP packets
- -u/--udp
  - capture only UDP packets
- --arp
  - capture only ARP frames, output the differs from others, more at 2.7
- --icmp
  - capture only ICMP packets
- -n
  - specifies how many packets/frame to capture, default value is 1

Due to the nature of this project (root privileges and network sniffing) it is expected of user to know what they are doing and not to mix parameters not compatible (e. g. ARP and port numbers). If parameters not compatible are given and program is stuck, it can be safely closed using

CTRL+C shortcut ("interrupt", SIGINT).

1.2 Program Capabilities

This program is able to capture traffic on Ethernet and Wi-Fi interfaces. It can sniff TCP, UDP and ICMP packets and ARP frames as well. For it to be able to capture network traffic it is required to run this program with administrator privileges.

List of program functions:

- choose interface
- list interfaces (--interface/-i given without value, not given at all)
- sniff TCP (--tcp/-t)
- sniff UDP (--udp/-u)
- sniff only on defined ports (-p)
- sniff given number of packets (-n) or 1 if n not given
- sniff ICMPv4 messages (--icmp) - do not have port numbers
- sniff ARP messages (--arp) - outputs MAC addresses instead of IP addresses
- combine more options
- write packet info
  - time as specified in RFC 3339 with milliseconds
  - source and destination IPv4 addresses
  - source and destination ports
  - frame length in bytes
- end program runtime using ctrl+c with safe memory handling
- can sniff these interfaces: Ethetnet, WiFi, DLT_LINUX_SLL, DLT_SLIP, DLT_PPP, DLT_RAW

1.3 Program Limitations

Program is limited to capture IPv4 traffic only. This decision has been made due to the capabilities of authors own home network for testing.

Program will not end itself when bad combination of protocol and port argument is selected. It is needed to exit the program using CTRL+C shortcut. More on this at 1.1.

2. Implementation

Program source code is divided to two files located in root folder. These files are `ipk-sniffer.cpp` and `ipk-sniffer.h`. File with `.cpp` extension contains implementation and program logic whereas `ipk-sniffer.h` is home for data structures, function definitions and include directives. Both files are heavily commented. Functions as whole are more described in `.h` file. Comments in `.cpp` file are trying to explain logic behind program structure.

There is also the README file containing brief description of program and its capabilities. Licenses for used code are also written there.

2.1 Function Main

Function Main is first to be execute after program starts. It does not compute anything itself given its function as *hub* of sorts for other functions which are sequentially called from it. These are: ArgCheck(), WriteInterface(), CheckInterface(), CreateSocket(), Sniff() and MrProper() named after brand of cleaning products, symbolizing its function.

It handles errors and creates signal handlers before it gives a green light to start sniffing.

2.2 Function ArgCheck

This function handles parsing and handling each of given program arguments. It uses `getopt` and `getopt_long` for this purpose. It has been written using Linux manual page example [1]. For it purpose it uses while loop and nested switch statement. Data from parsed arguments are stored in `params_data` structure. It also builds up string which is later used as PCAP filter used to choose right traffic to capture. It has to be differentiated while building this string, whether its element is first in string or not due to syntax rules (example: *tcp* at start of the string, *or tcp* in middle and at the end of the string).

Main difficulty was to make -i/--interface argument work with and without parameter. Its functionality has been achieved with a bit of code from [2].

2.3 Function WriteInterface

This is called after argument checking only if -i/--interface argument has not been given at all or it has been given without parameter. It is the first place in code where PCAP library is used. Function in question is `pcap_findalldevs` specifically. Here it is used to find information about all network devices in host. It is later cycled through every of these devices and their info is written out to the standard output. PCAP list is then safely closed and program is terminated.

2.4 Function CheckInterface

Determines if device name submitted in form of -i/--interface argument exists and terminates if the device in question is invalid. `pcap_findalldevs` function has been used again here.

2.5 Function CreateSocket

Goal of this function is to create functional PCAP device which is later used for frame capturing. First it opens the given device for capturing and saves its IP address and network mask for future use. It then creates PCAP filter from string created in 2.2 and applies it to the freshly created PCAP device.

2.6 Function Sniff

This is the heart of the ipk-sniffer. It is called after CreateSocket creates PCAP device and system signal handlers are setup to capture signals for program terminating.

First it determines the length of L2 header which is later used to get the right offset to access the right data in frame/packet. This header length varies between used protocol/medium. It is 14 bytes for Ethernet and 24 for Wi-Fi for example. These numbers are taken from [3].

PCAP loop is then initiated and run as many times as specified via -*n* argument. `pcap_loop` uses `doPacket` function to do its job of processing packets.

2.7 Function doPacket

DoPacket is called every time program reaches PCAP loop in function Sniff (2.6). First it gets current time in RFC3339 format with milliseconds in string type from function getCurrentTimeRFC3339 (2.9).

If ARP is set to be captured, it reads source and destination MAC addresses and prints main data line. It shows MAC addressees instead of IP addresses because ARP is working on L2 so it does not work with IP addressees. First main output line is outputted, then `print_payload` (2.10) is called to print frame data. Here I used `snprintf` function to store `char` MAC address data in std::string for easier output.

If ARP is not selected it uses value of L2 header from 2.6 and sets pointer after this section so L3 header can be read. From now accessible L3 header IP addresses and port numbers can be read. Protocol used is also read from this header and next procedure is decided on its value because TCP, UDP and ICMP use different headers it is important to differentiate between them to get the right data and offsets.

All is then set to be outputted. Main data line is written out first, then print_payload (2.10) handles writing out the rest of data.

2.8 Function MrProper

Named after famous brand of cleaning product this function does what its name suggests. It frees all PCAP resources and cleans after itself. Then it safely terminates the program.

2.9 Function getCurrentTimeRFC3339

This function has been taken from [4] and then modified to write its data to std::string and add milliseconds which were missing in original solution. [5] has also been heavily consulted in making of this program as a whole.

2.10 Functions print_hex_ascii_line and print_payload

These two functions handle the output of frame data in HEX and ASCII forms. I have completely taken these functions from [6]. No my modifications have been made. They are taken from licensed program which license can be found in README or ipk-sniffer.h.

3. Testing

I tested my implementation locally on my machine using comparison method. Two outputs to compare were 1) *ipk-sniffer output* and 2) *Wireshark output.*

3.1 Methodology

Comparison data has been generated by original traffic originating from or delivered to my own PC running Kubuntu 20.04. First I opened *Wireshark* as superuser and started capturing data from Ethernet interface. Immediately after that I ran my *ipk-sniffer* set up to capture 20 frames. After the capturing has been done I tried to locate same frames manually and compare them. For testing TCP, UDP and ARP data I used normal network traffic. For capturing ICMP messages I had to run ping utility originating at my PC with destination address of my LANs default gateway. This method helped me with setting the right offset for writing out HEX and ASCII data, because I could a) repeat the proccess and b) visually see differences in output and compensate for them.

3.2 Testing Examples

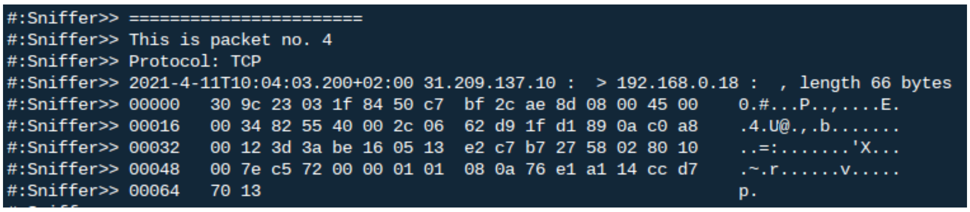

Figure 1: TCP packet captured with ipk-sniffer

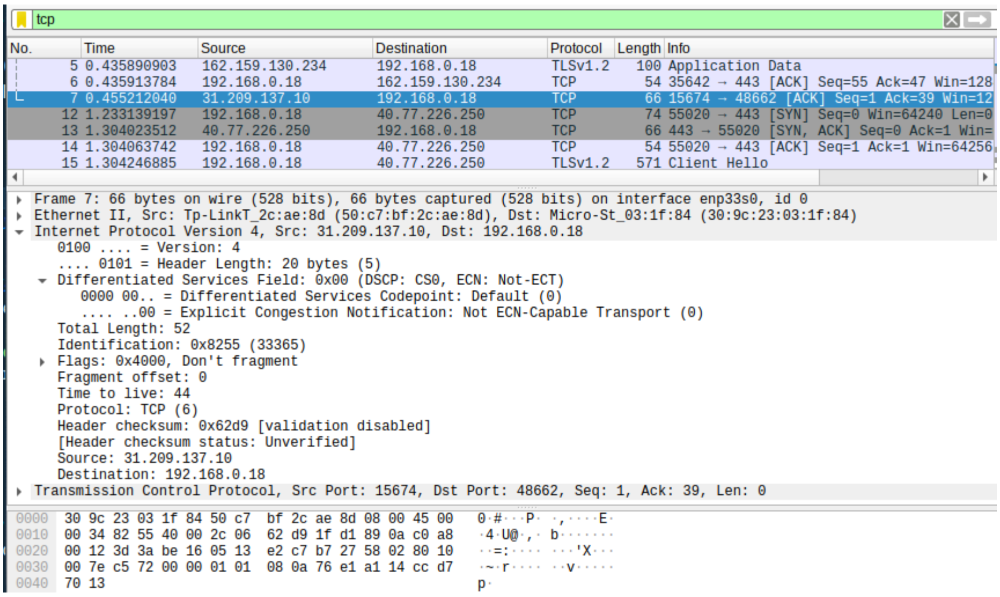

Figure 2: Same TCP packet captured with Wireshark

```
#:Sniffer>> ======================
#:Sniffer>> This is packet no. 5
#:Sniffer>> Protocol: UDP
#:Sniffer>> 2021-4-11T10:15:15.153+02:00 192.168.0.1 : 35239 > 224.0.0.251 : 5353 , length 85 bytes
#:Sniffer>> 00000   01 00 5e 00 00 fb 50 c7  bf 2c ae 8d 08 00 45 00    ..^...P..,....E.
#:Sniffer>> 00016   00 47 00 00 40 00 01 11  d8 01 c0 a8 00 01 e0 00    .G..@...........
#:Sniffer>> 00032   00 fb 89 a7 14 e9 00 33  7c d6 4d 36 00 00 00 01    .......3|.M6....
#:Sniffer>> 00048   00 00 00 00 00 00 03 31  39 32 03 31 36 38 01 30    .......192.168.0
#:Sniffer>> 00064   02 35 30 07 69 6e 2d 61  64 64 72 04 61 72 70 61    .50.in-addr.arpa
#:Sniffer>> 00080   00 00 0c 00 01                                      .....
```

Figure 3: UDP packet captured with ipk-sniffer

```
udp
No.  Time         Source         Destination    Protocol  Length Info
  35 1.645468177  192.168.0.1    224.0.0.251    MDNS        86 Standard query 0x4d32 PTR 192.168.0.150.in-addr.
  36 1.667041219  192.168.0.1    224.0.0.251    MDNS        85 Standard query 0x4d33 PTR 192.168.0.52.in-addr.a
  37 1.677314796  192.168.0.1    224.0.0.251    MDNS        85 Standard query 0x4d34 PTR 192.168.0.52.in-addr.a
  38 1.699019169  192.168.0.1    224.0.0.251    MDNS        85 Standard query 0x4d35 PTR 192.168.0.50.in-addr.a
  39 1.709518886  192.168.0.1    224.0.0.251    MDNS        85 Standard query 0x4d36 PTR 192.168.0.50.in-addr.a
  40 1.992441214  192.168.0.131  192.168.0.255  UDP        305 54915 → 54915 Len=263

Frame 39: 85 bytes on wire (680 bits), 85 bytes captured (680 bits) on interface enp33s0, id 0
Ethernet II, Src: Tp-LinkT_2c:ae:8d (50:c7:bf:2c:ae:8d), Dst: IPv4mcast_fb (01:00:5e:00:00:fb)
Internet Protocol Version 4, Src: 192.168.0.1, Dst: 224.0.0.251
User Datagram Protocol, Src Port: 35239, Dst Port: 5353
Multicast Domain Name System (query)

0000  01 00 5e 00 00 fb 50 c7  bf 2c ae 8d 08 00 45 00   ··^···P· ·,····E·
0010  00 47 00 00 40 00 01 11  d8 01 c0 a8 00 01 e0 00   ·G··@··· ········
0020  00 fb 89 a7 14 e9 00 33  7c d6 4d 36 00 00 00 01   ·······3 |·M6····
0030  00 00 00 00 00 00 03 31  39 32 03 31 36 38 01 30   ·······1 92·168·0
0040  02 35 30 07 69 6e 2d 61  64 64 72 04 61 72 70 61   ·50·in-a ddr·arpa
0050  00 00 0c 00 01                                     ·····
```

Figure 4: Same UDP packet captured with Wireshark

```
#:Sniffer>> ======================
#:Sniffer>> This is packet no. 4
#:Sniffer>> Protocol: ICMP
#:Sniffer>> 2021-4-11T09:50:58.940+02:00 192.168.0.18 : 2048 > 192.168.0.1 : 1118 , length 98 bytes
#:Sniffer>> 00000   50 c7 bf 2c ae 8d 30 9c  23 03 1f 84 08 00 45 00    P..,..0.#.....E.
#:Sniffer>> 00016   00 54 dc 47 40 00 40 01  dc fd c0 a8 00 12 c0 a8    .T.G@.@.........
#:Sniffer>> 00032   00 01 08 00 04 5e 00 02  01 93 62 aa 72 60 00 00    .....^....b.r`..
#:Sniffer>> 00048   00 00 5e 2f 00 00 00 00  00 00 10 11 12 13 14 15    ..^/............
#:Sniffer>> 00064   16 17 18 19 1a 1b 1c 1d  1e 1f 20 21 22 23 24 25    .......... !"#$%
#:Sniffer>> 00080   26 27 28 29 2a 2b 2c 2d  2e 2f 30 31 32 33 34 35    &'()*+,-./012345
#:Sniffer>> 00096   36 37                                               67
```

Figure 5: ICMP packet captured with ipk-sniffer

```
No.   Time         Source        Destination   Protocol  Length Info
  458 0.505727264  192.168.0.18  192.168.0.1   ICMP        98 Echo (ping) request  id=0x0001, seq=3/7
  459 0.505894719  192.168.0.1   192.168.0.18  ICMP        98 Echo (ping) reply    id=0x0001, seq=3/7
 1287 1.529745593  192.168.0.18  192.168.0.1   ICMP        98 Echo (ping) request  id=0x0001, seq=4/1
 1288 1.529934728  192.168.0.1   192.168.0.18  ICMP        98 Echo (ping) reply    id=0x0001, seq=4/1
 2134 2.553738923  192.168.0.18  192.168.0.1   ICMP        98 Echo (ping) request  id=0x0001, seq=5/1
 2135 2.553944469  192.168.0.1   192.168.0.18  ICMP        98 Echo (ping) reply    id=0x0001, seq=5/1
 2838 3.577729210  192.168.0.18  192.168.0.1   ICMP        98 Echo (ping) request  id=0x0001, seq=6/1

Frame 458: 98 bytes on wire (784 bits), 98 bytes captured (784 bits) on interface enp33s0, id 0
Ethernet II, Src: Micro-St_03:1f:84 (30:9c:23:03:1f:84), Dst: Tp-LinkT_2c:ae:8d (50:c7:bf:2c:ae:8d)
Internet Protocol Version 4, Src: 192.168.0.18, Dst: 192.168.0.1
Internet Control Message Protocol

0000  50 c7 bf 2c ae 8d 30 9c  23 03 1f 84 08 00 45 00   P··,··0· #·····E·
0010  00 54 e8 71 40 00 40 01  d0 d3 c0 a8 00 12 c0 a8   ·T·q@·@· ········
0020  00 01 08 00 2c 74 00 01  00 03 46 a8 72 60 00 00   ····,t·· ··F·r`··
0030  00 00 53 ac 00 00 00 00  00 00 10 11 12 13 14 15   ··S····· ········
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 6: Same ICMP packet captured with Wireshark

```
janz@honzalinux:~$ ping 192.168.0.1
PING 192.168.0.1 (192.168.0.1) 56(84) bytes of data.
64 bytes from 192.168.0.1: icmp_seq=1 ttl=64 time=0.274 ms
64 bytes from 192.168.0.1: icmp_seq=2 ttl=64 time=0.208 ms
64 bytes from 192.168.0.1: icmp_seq=3 ttl=64 time=0.186 ms
64 bytes from 192.168.0.1: icmp_seq=4 ttl=64 time=0.204 ms
64 bytes from 192.168.0.1: icmp_seq=5 ttl=64 time=0.219 ms
64 bytes from 192.168.0.1: icmp_seq=6 ttl=64 time=0.211 ms
64 bytes from 192.168.0.1: icmp_seq=7 ttl=64 time=0.253 ms
64 bytes from 192.168.0.1: icmp_seq=8 ttl=64 time=0.198 ms
64 bytes from 192.168.0.1: icmp_seq=9 ttl=64 time=0.225 ms
64 bytes from 192.168.0.1: icmp_seq=10 ttl=64 time=0.210 ms
64 bytes from 192.168.0.1: icmp_seq=11 ttl=64 time=0.206 ms
^C
--- 192.168.0.1 ping statistics ---
11 packets transmitted, 11 received, 0% packet loss, time 10234ms
rtt min/avg/max/mdev = 0.186/0.217/0.274/0.024 ms
```

Figure 7: Ping used to generate ICMP traffic

```
#:Sniffer>> 2021-4-11T09:56:29.372+02:00 50.C7.BF.2C.AE.8D. :  > 00:00:00:00:00:00: :  , length 60 bytes
#:Sniffer>> 00000   30 9c 23 03 1f 84 50 c7  bf 2c ae 8d 08 06 00 01    0.#...P..,......
#:Sniffer>> 00016   08 00 06 04 00 01 50 c7  bf 2c ae 8d c0 a8 00 01    ......P..,......
#:Sniffer>> 00032   00 00 00 00 00 00 c0 a8  00 12 00 00 00 00 00 00    ................
#:Sniffer>> 00048   00 00 00 00 00 00 00 00  00 00 00 00                ............
```

Figure 8: ARP frame captured with ipk-sniffer - notice MAC addresses instead of IP addresses

No.	Time	Source	Destination	Protocol	Length	Info
3403	4.766747218	Tp-LinkT_2c:ae:8d	Micro-St_03:1f:84	ARP	60	Who has 192.168.0.18? Tell 192.168.0.1
3404	4.766756328	Micro-St_03:1f:84	Tp-LinkT_2c:ae:8d	ARP	42	192.168.0.18 is at 30:9c:23:03:1f:84

```
Frame 3403: 60 bytes on wire (480 bits), 60 bytes captured (480 bits) on interface enp33s0, id 0
Ethernet II, Src: Tp-LinkT_2c:ae:8d (50:c7:bf:2c:ae:8d), Dst: Micro-St_03:1f:84 (30:9c:23:03:1f:84)
Address Resolution Protocol (request)
```

```
0000  30 9c 23 03 1f 84 50 c7  bf 2c ae 8d 08 06 00 01   0·#···P· ·,······
0010  08 00 06 04 00 01 50 c7  bf 2c ae 8d c0 a8 00 01   ······P· ·,······
0020  00 00 00 00 00 00 c0 a8  00 12 00 00 00 00 00 00   ········ ········
0030  00 00 00 00 00 00 00 00  00 00 00 00               ········ ····
```

Figure 9: Same ARP frame captured with Wireshark

4. Conclusion

This project has given me a lot of knowledge of working with network structures in C programming language as it was my first time using these capabilities of PCAP library. It has given me a look into working with in my opinion outdated and not so greatly made library documentation. This project helped me to understand the difference in multiple programming languages and their intended use and their workflow. For example now I know that Python (which syntax I truly hate with my whole hearth) which I used to make first IPK project is made to be more user friendly does not provide complete control like C/C++ with its pointer arithmetic.

Main takeaway from this project is the realization that protocol structured listed in many sources are real and internal data representation of these protocol really have the same data fields an length etc.

5. Bibliography

[1] getopt(3) - Linux manual page. 2021. URL https://man7.org/linux/man-pages/man3/getopt.3.html

[2] haystack: c - getopt does not parse optional arguments to parameters - Stack Overflow. 2009. URL https://stackoverflow.com/questions/1052746/getopt-does-not-parse-optional-arguments-to-parameters

[3] Link-Layer Header Types — TCPDUMP/LIBPCAP public repository. 2021. URL https://www.tcpdump.org/linktypes.html

[4] rfc3339.c GitHub. 2021. URL https://gist.github.com/jedisct1/b7812ae9b4850e0053a21c922ed3e9dc

[5] Klyne, G.: Date and Time on the Internet: Timestamps. RFC 3339, RFC Editor, 7 2002. URL https://tools.ietf.org/html/rfc3339

[6] The Tcpdump Group: tcpdump-htdocs/sniffex.c at master·the-tcpdump-group/tcpdump-htdocs·GitHub. 2021. URL https://github.com/the-tcpdump-group/tcpdump-htdocs/blob/master/sniffex.c

6. Table of Figures