Introduction

The goal of the project was to create the Ethernet packet sniffer. Sniffer should be able to analyse input and output traffic on a given interface, and to filter certain types of packets. The program supports packet detection of these protocols:

- IPv4:
  - ICP
  - UDP
  - ICMPv4
- IPv6:
  - ICP
  - UDP
  - ICMPv6
- ARP

Implementation details

General

The program utilises `pcap` and `netinet` libraries, the former for packet capturing and the latter for convenient type-casting of packet's header structures on octet sequences.
Entire program is made in OOP manner (except for the packet capture function – it can't be passed as an argument to the `pcap_loop` function). The `sniffer class` interface is stored in `sniffer.hpp` file, methods' implementation is stored in `sniffer.cpp` file.
The `Sniffer class` encapsulates all the required parameters such as strings for filter expression and interface, error buffer for `pcap` functions, integers for packet enumeration and packet's count limitation.

Command line argument parsing

The program uses `getopt_long` function with predefined arrays for both short and long options for command line arguments parsing. The string of filter expression is "constructed" during the arguments parsing: each new filter option (either packet types or port number) is added to result filter expression with `'or'` (in case of packet type) or `'and'` (in case of port number) between them.

Setting up the filter

Upon finishing of parsing, the program initialises `pcap` library, opens specified device, then it compiles and sets filter according to filter expression. If there is no interface specified program finds first device available via `pcap_findalldevs` function. If neither packet filters nor port are specified, then sniffer will listen on all ports to packets of all protocols.

Packet capturing

After everything is set, program starts sniffing packets inside `pcap_loop` via `packetHandler` function. Firstly, it defines variables for packet parsing, such as integers for length of packet, length of packet's part present, length of header offset for proper header type-casting; initialises variables for source and destination ports and defines Ethernet header by casting `ether_header*` structure on byte sequence in `packet` function argument.

Then the program defines timestamp by creating two string buffers (first for day-time, second for timezone) and integer for milliseconds. The strings are created by passing product of `localtime` function with corresponding options to `strftime` function, the milliseconds variable is a product of `lrint` function. All source data (seconds and milliseconds since the *Epoch*) is obtained from `header` of `pcap_pkthdr` type argument at the first place.

MAC source and destination addresses are also given by `header` argument's attributes. They are formatted (`\d\d:\d\d:\d\d:\d\d:\d\d:\d\d`) and passed to according variables by `sprintf` function.

Then function prints acquired information and proceeds in packet processing.
Afterwards, depending on ether type, function branches into three cases (IPv4, IPv6, ARP ether types), obtains source and destination IPs from corresponding ether type header and passes them to `ntohs` function, increments length of header offset and prints ports.

After that (except for ARP case) it branches again on 3 different cases (TCP, UDP, ICMPv4/6), gets source and destination ports and prints them.

At the end function prints out octet representation of entire packet via nested loop with the step of 16.
It prints 16 octet per line in such format:
<hexadecimal offset>: <hexadecimal octets' representation> <ascii octets' representation>

Output format

```
Packet #2
timestamp: 2022-04-16T19:54:17.884+02:00
src MAC: 94:3f:c2:07:ca:1a
dst MAC: ff:ff:ff:ff:ff:ff
frame length: 60
ether type: ARP
src IP: 147.229.212.1
dst IP: 147.229.214.40
0x0000:  ff ff ff ff ff ff 94 3f  c2 07 ca 1a 08 06 00 01  .......? ........
0x0010:  08 00 06 04 00 01 94 3f  c2 07 ca 1a 93 e5 d4 01  .......? ........
0x0020:  00 00 00 00 00 00 93 e5  d6 28 00 00 00 00 00 00  ........ .(......
0x0030:  00 00 00 00 00 00 00 00  00 00 00 00              ........ ....
```

Figure 1: ARP packet

```
Packet #1
timestamp: 2022-04-16T19:50:51.860+02:00
src MAC: b4:b6:86:dd:2e:39
dst MAC: ff:ff:ff:ff:ff:ff
frame length: 60
ether type: IPv4
src IP: 147.229.212.183
dst IP: 255.255.255.255
protocol: UDP
src port: 56056
dst port: 3956
0x0000:  ff ff ff ff ff ff b4 b6  86 dd 2e 39 08 00 45 00  ........ ...9..E.
0x0010:  00 24 cb ed 00 00 80 11  06 3f 93 e5 d4 b7 ff ff  .$...... .?......
0x0020:  ff ff da f8 0f 74 00 10  6a c0 42 01 00 02 00 00  .....t.. j.B.....
0x0030:  00 01 00 00 00 00 00 00  00 00 00 00              ........ ....
```

Figure 2: UDP packet

Testing

Testing part consisted of comparing the results of the program and the results of wireshark software.

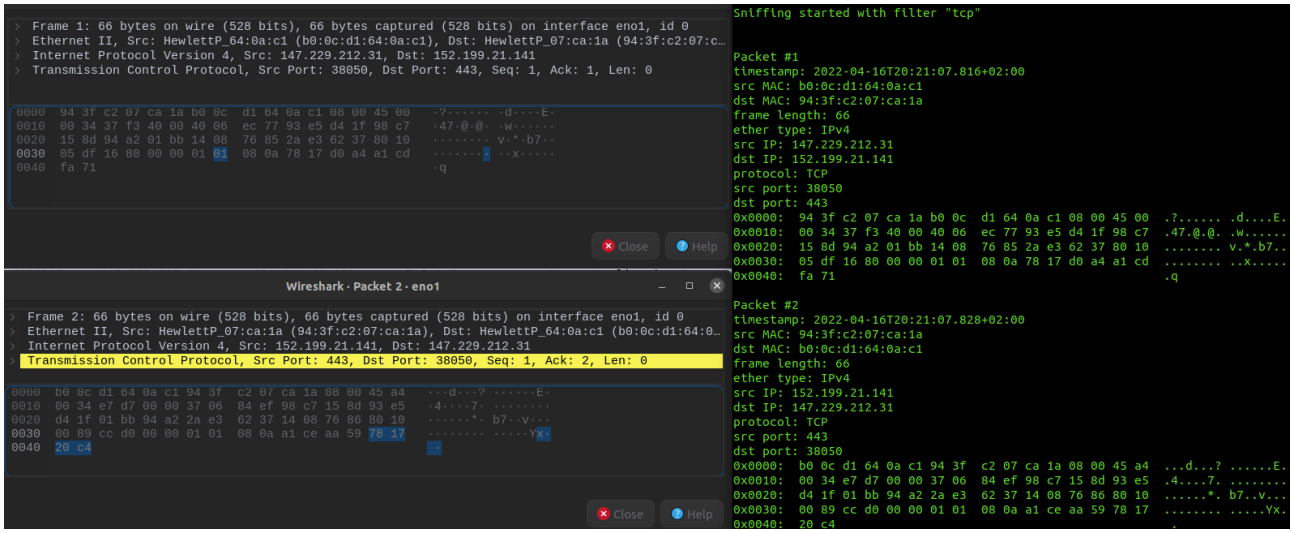

Figure 3: TCP packets comparison

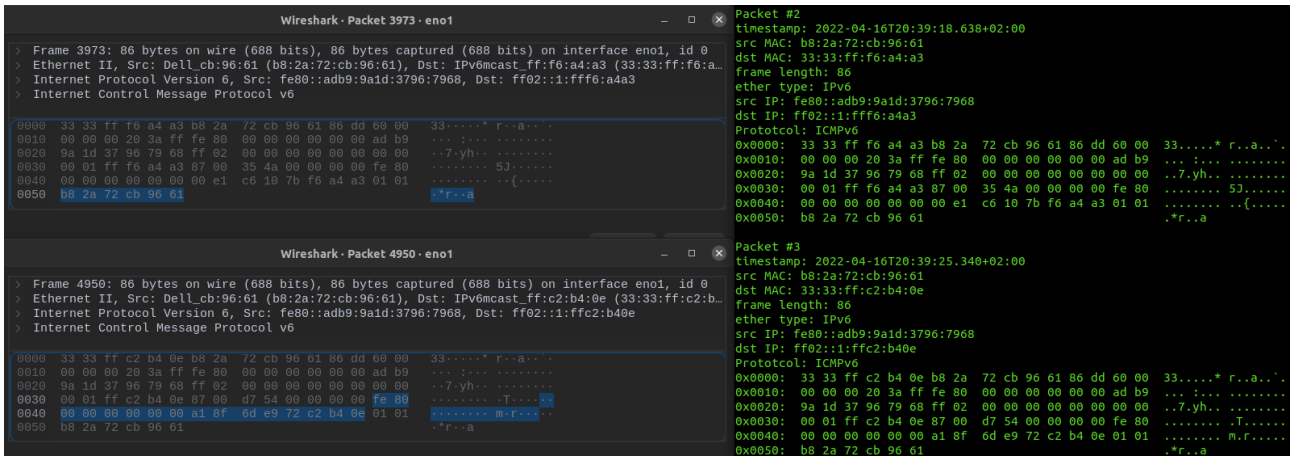

Figure 4: ICMPv6 packets comparison

References

[1] *Size of packets, type-casting.*
https://www.cs.dartmouth.edu/~sergey/cs60/lab4/tcp-listen.c

[2] *Printing octet representation of packets.*
https://www.binarytides.com/packet-sniffer-code-c-libpcap-linux-sockets/

[3] *Timestamp formatting.*
https://stackoverflow.com/questions/3673226/how-to-print-time-in-format-2009-08-1

[4] *Libpcap functions*
https://stackoverflow.com/questions/3673226/how-to-print-time-in-format-2009-08-1

[5] *Pcap functions usage code example.*
https://www.tcpdump.org/pcap.html

[6] *Ethernet frame info.*
https://en.wikipedia.org/wiki/Ethernet_frame