A simple packet sniffer

Patrik Skaloš

February 2022

Contents

1 Introduction

1.1 Packet sniffing

Packet sniffing is an act of capturing frames that are being sent from or received by a device on a certain network. Our program is able to capture *ethernet frames* on a single interface specified by the user. After a frame has been captured, various information about it can be printed, including source and destination addresses, ports and even the encapsulated data (although it is often encrypted).

1.2 Protocols

Protocols will be mentioned many times in this document, but what are they? They are simply sets of standards (rules) by which, for example, data is encapsulated, transferred and so on. Without protocols, networking would be a total chaos and thanks to them, it is very simple for devices to read any data received since they can rely on those rules.

1.2.1 OSI model

To explain the protocols supported by our packet sniffer, we must first introduce the OSI conceptual model of a networking system. Everything important to understand can be explained by figure 1.

	OSI Layer	TCP/IP	Datagrams are called
Software	**Layer 7** Application	HTTP, SMTP, IMAP, SNMP, POP3, FTP	Upper Layer Data
	Layer 6 Presentation	ASCII Characters, MPEG, SSL, TSL, Compression (Encryption & Decryption)	
	Layer 5 Session	NetBIOS, SAP, Handshaking connection	
	Layer 4 Transport	TCP, UDP	**Segment**
	Layer 3 Network	IPv4, IPv6, ICMP, IPSec, MPLS, ARP	**Packet**
Hardware	**Layer 2** Data Link	Ethernet, 802.1x, PPP, ATM, Fiber Channel, MPLS, FDDI, MAC Addresses	**Frame**
	Layer 1 Physical	Cables, Connectors, Hubs (DLS, RS232, 10BaseT, 100BaseTX, ISDN, T1)	**Bits**

Figure 1: OSI model [1]

1.2.2 Data units

Before any data can be sent over a network, it has to be formatted according to rules that various protocols provide. We need to understand differences between these data units:

- Frame: encapsulates a packet
- Packet: encapsulates a data segment

- Segment: encapsulates data

So, for example, sending a HTTP request from the application layer means dividing it into several portions if it is too big, encapsulating each portion to a segment, then to a packet and finally to a frame before sending it over a network media (e.g. cable) bit by bit.

Note: Encapsulation means taking some data and adding metadata around it (in networking, we use a header and only sometimes a trailer).

Note: Encapsulation doesn't always mean *data in segments in packets in frames*. Depending on a protocol, data, or sometimes even segments can be ommited (e.g. sending a packet with no user data in it since the real value of the packet is in the headers).

Note: It might be a bit confusing that a packet sniffer captures frames and not packets. I don't feel confident exaplaining why, but as I understand, word *packet* is often used when referring to frames or even segments and yes, to be precise, it should be called *frame sniffer* instead of *packet sniffer*.

1.3 Supported protocols

1.3.1 Data link layer protocols

Ethernet frame is a data link layer protocol data unit in which a packet is encapsulated along with MAC (physical) addresses, an *EtherType* field indicating which protocol is used for the packet and a *CRC* field, which is a sequence of bits used for error detection. See table 2 for visualisation. The payload field represents a single packet. An ethernet frame already contains all data necessary to travel over or to a different network and you could say it is a data unit of the highest level. Ethernet frame is the only data link layer protocol supported by our packet sniffer. For more information, see [3].

- **Header length**: 14 bytes (112 bits)
- **Trailer length**: 4 bytes (32 bits)
- **Important header fields**:

Field name	Data it represents	Field length	Field offset
Destination MAC address		48 bits	0 bits
Source MAC address		48 bits	48 bits
EtherType	Protocol used in the packet	16 bits	96 bits

Table 1: Important ethernet frame header fields

Destination MAC address	Source MAC address	EtherType	**Payload**	CRC

Table 2: Ethernet frame contents

1.3.2 Network layer protocols

IPv4 (internet protocol, version 4) is the most common one and is used to transfer data over the internet. For more information, see [5].

- **Header length**: 20 to 60 bytes (160 to 480 bits)
- **Important header fields**:

Field name	Data it represents	Field length	Field offset
IHL	Header length in 32-bit words	4 bits	4 bits
Protocol	Protocol of the encapsulated segment	8 bits	72 bits
Source IP address		32 bits	96 bits
Destination IP address		32 bits	128 bits

Table 3: Important IPv4 packet header fields

Note: We need to read the header length to know how many bytes to skip to get to the data, since IPv4 packet does not have a fixed header length.

Note: We don't need the version field as we already know the internet protocol version thanks to the *EtherType* field of the ethernet frame.

IPv6 (internet protocol, version 6) is very similar to IPv4. It is supposed to be an improvement because IPv6 provides much larger address space as it's addresses are 128 bits long, while IPv4 addresses are only 32 bits long. For more information, see [6].

- **Header length**: 40 bytes (320 bits)
- **Important header fields**:

Field name	Data it represents	Field length	Field offset
Next Header	equivalent to *Protocol* field of IPv4	8 bits	48 bits
Source IP address		128 bits	64 bits
Destination IP address		128 bits	192 bits

Table 4: Important IPv6 packet header fields

ICMP (internet control message protocol) is used by devices for diagnostics and reporting errors. According to my research, ICMP packet is not encapsulated in an IP packet in any way (evidence lies in the fact that it is a network layer protocol, not transport layer protocol. However, ICMP packet could be represented as an union of IP header, ICMP header and data, since the beginning of the ICMP header is the same as an IP header. For more information, see [4].

Note: Command *ping* uses ICMP.

- **Header length**: 8 bytes (32 bits)
- **Important header fields**:

Field name	Data it represents	Field length	Field offset
Type	Message type	8 bits	0 bits
Code	Message code (subtype)	8 bits	8 bits

Table 5: Important ICMP header fields (without the IP part)

ARP (address resolution protocol) is a protocol used by devices to keep track of IP addresses associated with MAC addresses of devices in a given network. These associations are kept in a so-called *ARP table*. The packet originates on the network layer itself. A single ARP packet can represent an ARP request or an ARP response. For more information, see [2].

- **Header length**: 8 bytes (64 bits)
- **Important packet fields**:

Field name	Data it represents	Field length	Field offset
OPCODE	Indicating a request or a reply	16 bits	48 bits
Source MAC address		48 bits	64 bits
Source protocol address	Sender IP address	32 bits	112 bits
Target MAC address		48 bits	144 bits
Target protocol address	Target IP address	32 bits	192 bits

Note: Protocol address lengths (and therefore offsets) might be inaccurate since different sources tell different numbers

Table 6: Important ARP packet fields

1.3.3 Transport layer protocols

TCP (transmission control protocol) is the most common transport layer protocol. It provides a reliable way to send and receive data over a network and it directly encapsulates a raw segment of data by adding a TCP header to it. The main aspects of the TCP are:

- Before any data is sent, a three-way handshake is used to initialize the network connection between two nodes (devices)
- If a packet is lost or damaged (data received does not match the data sent) on the way, it is retransmitted. This is to make sure that all packets arrive to the destination perfectly fine

This process is great, but as you can imagine, this increases the latency since the handshake and error-checking takes time and frames have to be bigger. For more information, see [7].

- **Header length**: 20 to 60 bytes (160 to 480 bits)
- **Important header fields**:

Field name	Data it represents	Field length	Field offset
Source port		16 bits	0 bits
Destination port		16 bits	16 bits
Sequence number		32 bits	32 bits
Acknowledgement number		32 bits	64 bits
Data offset	Header length in 32-bit words	4 bits	96 bits
Flags		8 bits	104 bits

Table 7: Important TCP segment header fields

UDP (user datagram protocol) on the other hand *does not care if the packets get damaged or lost* which increases the communication speed. No handshake is done and communication uses a simple *request-response* method. This protocol is however unsuitable for most applications and is used mainly for audio and video streaming (large files where speed matters and minor damages to packets are acceptable). For more information, see [8].

Note: UDP segments are also called datagrams.

- **Header length**: 8 bytes (64 bits)
- **Important header fields**:

Field name	Data it represents	Field length	Field offset
Source port		16 bits	0 bits
Destination port		16 bits	16 bits

Table 8: Important UDP segment header fields

2 Implementation

2.1 Networking libraries used

This is a list of networking libraries used and some of functions or structures they provide:

- `pcap.h` provides several important functions for discovering available interfaces and using them
- `arpa/inet.h` provides `inet_ntop` function used to convert IP addresses to strings, `ntohs` and `ntohl` functions to parse header data in case of different network and host `endianness`
- `netinet/ether.h` provides `ether_header` structure that an ethernet header can be typecasted to
- `netinet/ip6.h` provides `ip6_hdr` structure
- `netinet/tcp.h` provides `tcp_hdr` structure
- `netinet/ip_icmp.h` provides `ip` and `icmphdr` structures

2.2 Capabilities and limitations

Supported data link layer protocols:

- Ethernet (our sniffer only captures Ethernet frames)

Supported network layer protocols:

- IPv4
- IPv6
- ICMP
- ARP

Supported transport layer protocols:

- TCP
- UDP

2.3 Execution

```
./ipk-sniffer [-h] [-i int] [-p port] [--tcp] [--udp] [--arp] [--icmp]
    [-n num]

-h print this help
-i interface to sniff on
-p port to use
--tcp, --udp, --icmp, --arp select protocols to filter the traffic by.
  If none is used, all four protocols are selected by default
-n amount of packets to stop after

Note: Program may require root privileges
```

2.3.1 Examples

Print usage

```
./ipk-sniffer -h
```

Print all available interfaces

```
./ipk-sniffer -i
```

Listen on interface wlo1 for 4 TCP frames

```
./ipk-sniffer -i wlo1 --tcp -n 4
```

Listen on interface wlo1 for any one frame on port 80

```
./ipk-sniffer -i wlo1 -p 80
```

Listen on interface wlo1 for an ICMP frame or UDP frame on port 80

```
./ipk-sniffer -i wlo1 -p 80 --udp --icmp
```

2.4 Output format

2.4.1 Data printed about all captured frames:

```
timestamp:              YYYY-MM-DDThh:mm:ss.uuuuuuZ
src MAC:                xx:xx:xx:xx:xx:xx
dst MAC:                xx:xx:xx:xx:xx:xx
frame length:           N bytes
network layer proto:    PROTOCOL

WHOLE_FRAME_AS_BYTES_AND_CHARACTERS
```

Note: *DATA* is printed at the end of the report. For more information see 2.4.4

2.4.2 Additional data printed depending on the packet protocol

ARP:

```
operation:              request/reply/unknown: CODE
src MAC:                xx:xx:xx:xx:xx:xx
src IP:                 IP_ADDRESS
tgt MAC:                xx:xx:xx:xx:xx:xx
tgt IP:                 IP_ADDRESS
```

ICMP:

```
transport layer proto:   PROTOCOL
src IP:                  IP_ADDRESS
dst IP:                  IP_ADDRESS
msg type:                N
msg code:                N
```

IP:

```
transport layer proto:   PROTOCOL
src IP:                  IP_ADDRESS
dst IP:                  IP_ADDRESS
src port:                N
dst port:                N
```

2.4.3 Additional data printed depending on the segment protocol

TCP:

```
seq number:              N
ack number:              N
flags as bits:           bbbbbbbb
```

UDP: none

2.4.4 Data format

Data is printed at the end of a report of a captured frame and prints the entire frame as bytes and characters in the following format:

```
OFFSET_IN_HEX: 16_BYTES_IN_HEX_SEPARATED_BY_SPACES 16_BYTES_AS_CHARACTERS
```

Example:

```
0x0000: 17 03 03 00 39 64 b2 55  4f e8 4f ba 40 6c 97 32  ....9d.U O.O.@l.2
0x0010: 66 d1 cd 7b 11 3e 3c 13  0d 58 67 8a bc 77 ee 71  f..{.><. .Xg..w.q
0x0020: 31 b9 75 97 8e 47 1b 94  2d 8c cd de 22 51 13 81  1.u..G.. -..."Q..
0x0030: b1 8a 98 0b 7c a6 09 bb  ee 3d f3 17 80 d3        ....|... .=....
```

2.4.5 TCP segment output example

```
timestamp:               2022-03-01T08:18:59.210383Z
src MAC:                 00:05:96:1f:4e:30
dst MAC:                 ac:67:5d:1a:fa:89
frame length:            68 bytes
network layer proto:     IPv4
transport layer proto:   TCP
src IP:                  168.119.79.28
dst IP:                  147.299.189.109
```

```
src port:               1
dst port:               208
seq number (raw):       427943615
ack number (raw):       3390744016
flags as bits:          00011000

0x0000: ac 67 5d 4f fa 83 00 04  96 1d 4e 30 08 00 45 a4  .g]O.... ..N0..E.
0x0010: 00 34 42 6b 40 00 38 06  5e b0 97 65 c1 45 93 e5  .4Bk@.8. ^..e.E..
0x0020: b4 74 01 bb e5 46 bb 9a  09 94 74 f6 da cb 80 10  .t...F.. ..t.....
0x0030: 01 25 48 84 00 00 01 01  08 0a fb 5d d5 82 b3 1a  .%H..... ...]....
0x0040: 0c 21 00 ff                                       .!..
```

2.5 Notes on my implementation

- Since the `getopt.h` library isn't very supportive of long options with a value (for example `--intefrace name`), I had to give up checking the `opterr` variable to see if any errors occured while parsing the options. However, I don't see this being that big of a deal.
- When printing data, I print bytes until the end of the frame indicated by `caplen` variable contained in a `pcap_pkthdr` structure returned by `pcap_next` function (this function is the one that captures frames and returns the mentioned header and a pointer to the frame as a sequence of bytes). The problem is that I couldn't find any information about whether the `caplen` is calculated including the `CRC` field (Ethernet frame trailer) length or not. I assume that reading all the way to `caplen` is the right way since I tested it and often received meaningful printable characters as the last bytes. This, however, does not make much sense and at first, I was certain I should trim the last 4 bytes (`CRC` length).
- To provide a safe way to kill the server using the `SIGINT` signal at any time (e.g. using `Control+C`), we are capturing said signal and before exiting, *freeing* all resources.
- I understand that addresses and such contained in a frame can be encoded using a different *endianness* than our computers use and they often need to be converted. Luckily, `arpa/inet.h` provides `ntohs` and `ntohl` functions to convert a short integer (16 bits) or a long integer (32 bits) from the network *endianness* to the host *endianness*. However, by testing I found out it should not be applied to **all** the header fields longer than 2 bytes. So the question has to be asked: when exactly should I use the `ntohs` or `ntohl` functions to read data from a frame? The closest thing to an answer was to use it on any header field with length bigger than a byte, which is, obviously, not specific enough. The way I handled this issue was using said functions everywhere and testing—in case the program was printing incorrect values, I removed the converting function call and tested again to see if it helped.
- There is a function called `inet_ntoa` in the `arpa/inet.h` library which can be used to convert a MAC address to a string. This was however not suitable as it trimmed any leading zeros and prints `0:0:3:ab...` instead of `00:00:03:ab...`. Luckily, there is a simple (although a bit ugly ugly) way to print it without this function in just four lines (although, it uses `printf` which is not very *C++-like*).

2.6 Testing

2.6.1 Testing on Ubuntu 20.04.4 LTS - personal computer

First tests were performed on my personal computer and while I initially tested using commands like `curl`, `ping` and such, the real testing only began after I installed and opened `wireshark`. I tried all possible combinations of supported protocols and compared my outputs to the `wireshark`'s outputs. It was a bit tricky to find the pair to compare a single frame with but it was totally worth it. Although, I was able to isolate TCP packets by using a simple HTTP server on a custom port. Doing that, I found several mistakes as displaying ports, IPv6 addresses and even the timestamps (I was displaying microseconds in a hexadecimal format).

2.6.2 Testing on Ubuntu 20.04.2 LTS - reference virtual machine

I didn't expect that testing on the reference virtual machine would be unsuccessful due to it being pretty much the same operating system as on my personal computer, but I tested there anyways. I tested ARP, ICMP, TCP and UDP protocols but could not test IPv6 capabilities, because the virtual machine (or `VirtualBox` or whatever else) does not support it. Anyways, all tests were successful.

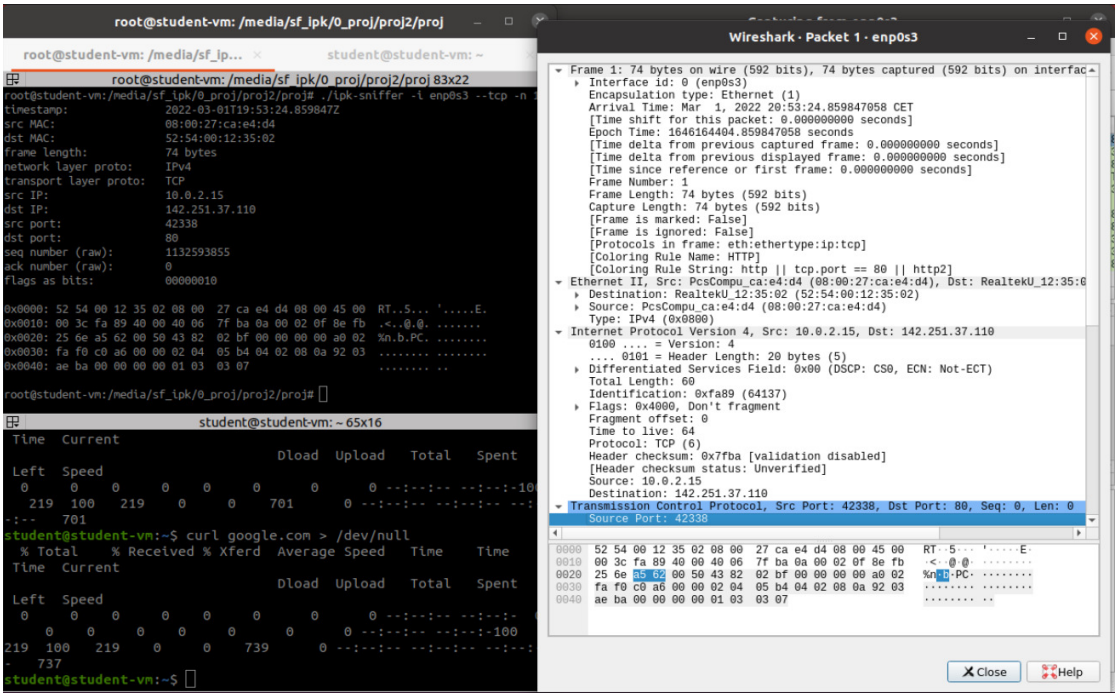

Figure 2: Testing on *Ubuntu 20.04.2 LTS* using *Wireshark*

3 Conclusion

Programming a packet sniffer in C++ is not as hard as I initially thought thanks to all the libraries that help us parse datagram headers. The most important things are to understand the OSI model layers and protocols and datagrams associated with those layers, to not forget to convert from the network *endianness* using `htons` and `htonl` and to know about the available networking libraries that save us a ton of work. That is not to say that the project was easy. Understanding the layers and protocols took a lot of time and some things still confuse me or I am still not sure about (e.g. what exactly is the relationship between the ICMP and IP protocols).

References

[1] "fadil". Osi models, 2018. [Online; accessed 01.03.2022; avilable at https://techsoftcenter.com/osi-model].

[2] Wikipedia. Address resolution protocol, 2022. [Online; accessed 02.03.2022; available at https://en.wikipedia.org/wiki/Address_Resolution_Protocol].

[3] Wikipedia. Ethernet frame, 2022. [Online; accessed 02.03.2022; available at https://en.wikipedia.org/wiki/Ethernet_frame].

[4] Wikipedia. Internet control message protocol, 2022. [Online; accessed 02.03.2022; available at https://en.wikipedia.org/wiki/Internet_Control_Message_Protocol].

[5] Wikipedia. Ipv4, 2022. [Online; accessed 02.03.2022; available at https://en.wikipedia.org/wiki/IPv4].

[6] Wikipedia. Ipv6, 2022. [Online; accessed 02.03.2022; available at https://en.wikipedia.org/wiki/IPv6].

[7] Wikipedia. Transmission control protocol, 2022. [Online; accessed 02.03.2022; available at https://en.wikipedia.org/wiki/Transmission_Control_Protocol].

[8] Wikipedia. User datagram protocol, 2022. [Online; accessed 02.03.2022; available at https://en.wikipedia.org/wiki/User_Datagram_Protocol].