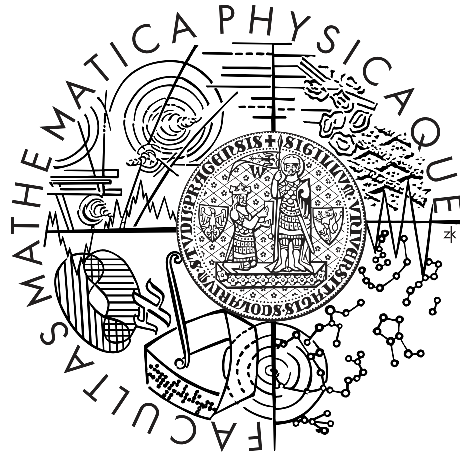
Charles University in Prague
Faculty of Mathematics and Physics

BACHELOR THESIS



Nataliia Korop

Packet capture for HelenOS

Department of Distributed and Dependable Systems

Supervisor of the bachelor thesis:	Mgr. Vojtěch Horký, Ph.D.
Study programme:	Computer Science
Specialization:	Programming and Software Development

Prague 2025

I thank my supervisor, Mgr. Vojtěch Horký, Ph.D., for countless explanations and advice during this work.

I would also like to thank the HelenOS team for their assistance on mailing lists and advice.

I declare that I carried out this bachelor thesis independently and only with the cited sources, literature, and other professional sources.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular the fact that the Charles University in Prague has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 paragraph 1 of the Copyright Act.

In Prague on .... Nataliia Korop

Title: Packet capture for HelenOS

Author: Nataliia Korop

Department: Department of Distributed and Dependable Systems

Supervisor of the bachelor thesis: Mgr. Vojtěch Horký, Ph.D., department of Distributed and Dependable Systems

Abstract: The goal of the thesis is to add support for traffic dumping on network drivers to HelenOS.

The text contains an introduction to the HelenOS system, an analysis of the relevant parts of the system, an analysis of the popular formats, and tools to dump traffic. The text analyzes different approaches in terms of efficiency, workload, and loading other components.

The work also describes a prototype implementation of the framework that adds support for dumping packets on all network drivers and describes how, with the help of the framework, the support to dump packets can be added to any driver in HelenOS.

Keywords: HelenOS, packet dumping, networking

Název práce: Packet capture for HelenOS

Autor: Nataliia Korop

Katedra: Katedra distribuovaných a spolehlivých systémů

Vedoucí bakalářské práce: Mgr. Vojtěch Horký, Ph.D., Katedra distribuovaných a spolehlivých systémů

Abstrakt: Cílem této práce je přidání podpory pro zachycení provozu na síti do HelenOSu.

Text obsahuje úvod do systému HelenOS, analýzu příslušných částí systému, analýzu oblíbených formátů a nástrojů pro zachycení provozu. Text analyzuje různé přístupy pro implementaci podpory z hlediska efektivity, náročnosti, a zatížení dalších komponent.

Práce také popisuje prototyp frameworku, který implementuje podporu pro výpis paketů na všech síťových driverech, a popisuje, jak lze pomocí frameworku přidat podporu pro zachycení paketů do obecného driveru v systému HelenOS.

Klíčová slova: HelenOS, packet dumping, networking

Contents

1 Introduction **3**
1.1 Goals . . . 3
1.2 Thesis structure . . . 3

2 Context HelenOS **5**
2.1 IPC . . . 5
2.2 Device drivers . . . 6
2.3 Network drivers . . . 6
2.4 TCP stack . . . 6
2.5 Applications . . . 6
2.6 Testing . . . 6

3 Analysis **8**
3.1 File format for storing packets . . . 8
3.2 PCAP format structure . . . 8
3.3 TCP stack in more details . . . 9
3.3.1 Link layer: ethip . . . 10
3.3.2 Network layer: inetsrv . . . 10
3.3.3 Transport layer: tcp/udp . . . 13
3.4 Architecture designs . . . 13
3.4.1 Forwarding server . . . 14
3.4.2 Dumping Server . . . 14
3.4.3 Application and driver . . . 15
3.4.4 Final decision . . . 16
3.5 Ideas for testing . . . 17

4 Implementation **18**
4.1 pcap library . . . 18
4.1.1 PCAP format for dumping . . . 18
4.1.2 Modification of the NIC code . . . 19
4.1.3 Packet dumper . . . 21
4.1.4 Discoverability of the driver . . . 24
4.1.5 Dump controlling . . . 27
4.2 Applications . . . 28
4.2.1 `pcapctl` . . . 28
4.2.2 `pcapcat` . . . 28

5 Benchmarking and testing 30
5.1 Testing . . . 30
5.1.1 Automated test scripts . . . 30
5.1.2 Host network . . . 31
5.2 Benchmarking . . . 33

6 Conclusion 37
6.1 Future work . . . 37

Bibliography 38

List of Figures 39

A Starting and running HelenOS 40
A.1 Building . . . 40
A.2 Running . . . 41
A.3 Running HelenOS with prebuilt image . . . 42

B User documentation for `pcapctl` 44
B.1 User documentation . . . 44
B.2 Examples of usage . . . 45

C User documentation for `pcapcat` 48

D Source code overview 49

E Running CI tests locally 50

F Programmer documentation 51

Chapter 1

Introduction

Network communication has always been complex. It takes effort and time to implement endpoint network drivers or implement TCP/IP stack. It takes even more effort and time to debug and find problems during the process of implementing. Therefore, helping tools such as network sniffers are designed. Such network sniffers capture packets being transmitted or received over a network to which the computer is attached. Network sniffers are designed to help developers analyze network traffic or use it for debugging purposes.

1.1 Goals

Currently, HelenOS does not have any support for network traffic capture. The main goal of the thesis is to implement a framework for network traffic capturing. As an inspiration, we used tcpdump (10), which is a command-line packet analyzer. tcpdump served only as an inspiration, as our goal is to implement native support for network analysis in HelenOS.

Our goals for the thesis:

- Our framework will be able to receive commands to start/stop dumping for different interfaces.
- To demonstrate the usability of our framework, we will add support for dumping packets on all network drivers in HelenOS.
- The implementation will be extensible for further development.
- We will design several tests for our framework.

1.2 Thesis structure

In Chapter 1, we introduced HelenOS and its essential components for the thesis and described the goals of the thesis.

In Chapter 3, we will analyze the approaches that were discussed during the work and the justification for the chosen approach.

In Chapter 4, we will provide a detailed description of the implementation.

In Chapter 5, we will show the results of the evaluation and testing of the final implementation.

In Chapter 6, we will summarize our work and discuss the results we have achieved and mention possible future extensions.

Chapter 2

Context HelenOS

HelenOS is a portable, microkernel-based, multiserver operating system designed and implemented from scratch. HelenOS is non-POSIX. (4)

Network communication in HelenOS travels through several different processes before arriving to the actual application, which is quite different from traditional monolithic kernels. The processes are NIC driver, ethernet, IP and TCP controllers. In this chapter we will describe these processes and other relevant components for the work.

2.1 IPC

HelenOS runs instances of user space programs as isolated tasks with their own address space. As a result, different tasks do not interfere with each other's address spaces. The kernel provides support for inter-process communication (IPC) for tasks to communicate with each other. Both synchronous and asynchronous communication is possible, but the IPC in HelenOS is mostly asynchronous.

HelenOS asynchronous framework represents a layer above the low-level IPC mechanism. The framework takes care of waiting for calls and redirecting them if necessary; therefore, a programmer can use asynchronous communication without deep understanding of implementation details.

IPC allows the sender to send several calls without getting an immediate answer. After sending a call, the sender does not need to wait for an answer to continue. Receiver will get the calls in the right order.

IPC provides functionality to pass short messages, pass large blocks of data, and share the memory.

Communication is carried out via asynchronous session. The session is initialized before any communication can take place. (5)

2.2 Device drivers

Drivers are represented as separate processes in HelenOS. HelenOS Device Driver Framework (DDF) takes care of managing the device topology graph (device tree), starting individual drivers, or attaching them to devices. DDF implements functionality that is useful for device drivers. The framework consists of a server, the Device Manager (`devman`), and the Device Driver library (`libdrv`). Every driver is linked to `libdrv`, which communicates with the Device Manager via IPC; the communication is abstracted from the driver.

Drivers, in the same way as other processes, can communicate with the rest of the world via IPC. (6)

2.3 Network drivers

Network drivers use a common framework, and there is another layer of abstraction in `libdrv`. The layer of abstraction is `nic` - Network Interface Controller. `nic` encapsulates specific driver and hides implementation details. We want to enable every implemented network driver to dump packets with least possible code modifications, therefore `nic` layer is a good place to implement dumping in.

More details about `nic` interface are in section 4.1.2.

2.4 TCP stack

The TCP stack in HelenOS is split into different servers. The structure partially mirrors network layers; high-level protocols often have their own servers. Details are described later in section 3.3.

2.5 Applications

Applications we are interested in are CLI applications. In HelenOS structure they reside under `uspace/app`.

Applications in HelenOS start with one IPC connection and later will acquire another if they need to communicate with other processes. Section 4.2 provides description of two applications that were implemented during the work.

2.6 Testing

HelenOS provides a framework for testing. The framework aims for unit testing and is represented as an application that users can run from inside of HelenOS.

Another way of testing is integration testing. HelenOS has a sub-project, Continuous Integration (CI). CI requires scenarios where commands and expected

results are written. CI project starts HelenOS automatically and runs commands from the scenarios.

We will use the second option as some of the tests will require different configuration of HelenOS and CI project enables user to set configuration before starting HelenOS.

Chapter 3

Analysis

In this chapter, we will explore file formats for storing packets and choose the format we will use in our implementation. We will consider several architectural approaches for the framework and select the most suitable one based on their advantages and disadvantages.

But before that, we will describe the TCP stack in more detail and give an example of the interaction of an application with the IPC stack. Thus, we will provide more context for the reader.

3.1 File format for storing packets

We considered several file formats: PCAP, CAP, PCAPNG.

CAP is a popular file format for storing raw data for further analysis. Among others, the format can be used for storing packets. We could not find the standard that describes the structure of the file format, therefore, we decided against it.

PCAPNG is a flexible successor to the PCAP. However, the format is not as widely used as PCAP.

PCAP format is the main capture file format among networking tools according to (11). It is a default option for tools such as tcpdump, nmop.

We chose the PCAP format for our prototype, as it has a simple structure and is easy to implement from scratch without using third-party libraries. (7)

3.2 PCAP format structure

The file starts with the file header, followed by zero or more packet records. There is a packet record for every packet. The file header is in the image 3.1. The file header length is 24 bytes.

Meaning of some nontrivial header fields:

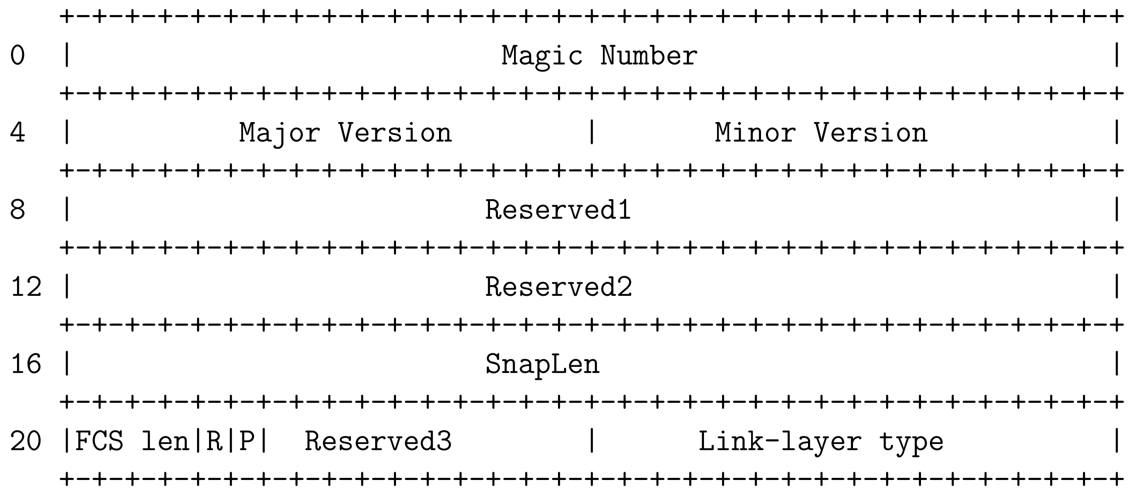


Figure 3.1: File header. See online reference (1).

- `Magic number`: define whether the timestamps are in microseconds or nanoseconds.
- `Snaplen`: maximum number of bytes that will be captured from every packet. If the number of bytes of the packet exceeds the snaplen, the snaplen bytes will be stored, and the rest will be ignored.
- `Linktype and additional information`: contains link layer type of the packet and may have additional information. Fig. 3.1.
  - `Link-layer type`: contains link-layer type for the packets in the file.
  - `P`: if set - Frame Check Sequence (FCS) value is present. If unset - there is no FCS.
  - `R`: must be set to 0 and ignored. Non-zero value is treated as an error.
  - `FCS length`: indicates the number of the 16-bits words of FCS that are appended to each packet, if the `P` is set. If the `P` is unset, and the `FCS length` can not be retrieved from the `Link-layer` type the `FCS length` is unknown.

After the header, the packet records follow with the structure illustrated in Fig. 3.2. (1)

3.3 TCP stack in more details

As was mentioned before, the TCP stack is split into several servers. They are all implemented in userspace. The only server that communicates directly with the hardware is the Network Interface Controller (`nic`). `nic` receives and sends frames (8).

Fig. 3.3 demonstrates data flow between services of the TCP stack in HelenOS. The division of work, captured by Figure 3.3, was best described by the

```
   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
0  |                      Timestamp (Seconds)                      |
   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
4  |            Timestamp (Microseconds or nanoseconds)            |
   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
8  |                    Captured Packet Length                     |
   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
12 |                    Original Packet Length                     |
   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
16 /                                                               /
   /                          Packet Data                          /
   /                  variable length, not padded                  /
   /                                                               /
```

Figure 3.2: Packet record. See online reference (1).

author of IPv6 support in HelenOS [see (8)]. Therefore sections 3.3.1, 3.3.2 and 3.3.3 cite the work.

Fig. 3.4 demonstrates path of the Ethernet frame from the driver that received the frame to the end user application (from bottom to top). In driver frame , in end user application represented as stream. (1)

3.3.1 Link layer: ethip

`Ethip` server implements Ethernet protocol. In the direction from bottom to top the server receives frames from the driver. In case of ARP frames server processes them itself, other frames are sent to `inetsrv` server. (8)

In the direction from top to bottom `ethip` receives frames from `inetsrv`, resolves MAC address (with the help of ARP protocol) and sends the packed datagrams. If the MAC address is not in ARP cache then before sending packed datagrams ARP request with MAC address is sent. Datagrams are not processed until reply is received. If the reply is not received under definite time datagrams are discarded.

Alternative to the Ethip is loopip server, that represents loopback interface.

3.3.2 Network layer: inetsrv

`Inetsrv` implements the IP protocol. The server communicates with `ethip` and `loopip` servers via IPC as a client and as a server provides services to upper layers. When `inetsrv` receives IP datagram from the `ethip` or `loopip` it processes the packet or sends it to the `tcp/udp` server based on the type of the datagram. As a server `inetsrv` provides three interfaces for IPC communication: `inet`, `inetcfg` and `inetping`. They are represented as IPC ports.

`Inet` is used by transport protocol servers `tcp` and `udp` to communicate with `inetsrv`. After connecting to `inetsrv` transport protocols register their protocol

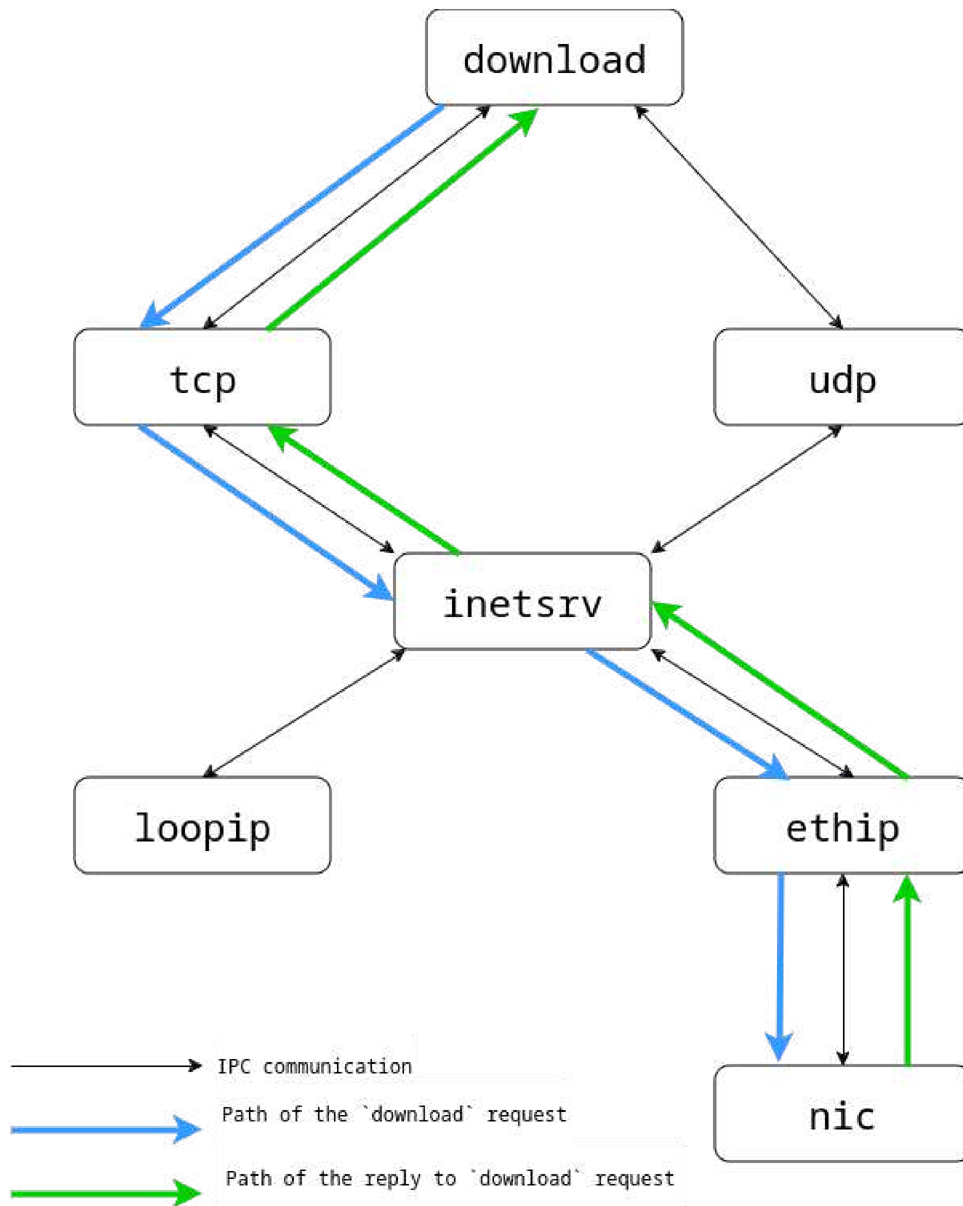


Figure 3.3: Processing of the ‘`download http://helenos.org/`‘ in context of TCP stack in HelenOS

numbers. The step is necessary for further correct delivery of incoming packets. When `inetsrv` receives an IP datagram, server checks protocol number. If the protocol number is ICMP, then server processes it itself. Otherwise it finds protocol numbers among registered ones, extracts packet from datagram and sends it to matching transport protocol server. If protocol number is not registered, then received datagram is discarded.

Transport protocol servers can ask for the source IP address in order to use it for a given destination IP address. `inetsrv` checks if the destination IP address is reachable directly from a link-layer interface. If the address is directly reachable, the server returns its own address in the same network. If the address is not directly reachable, then after consulting routing tables the server returns its own address in the network of the best router.

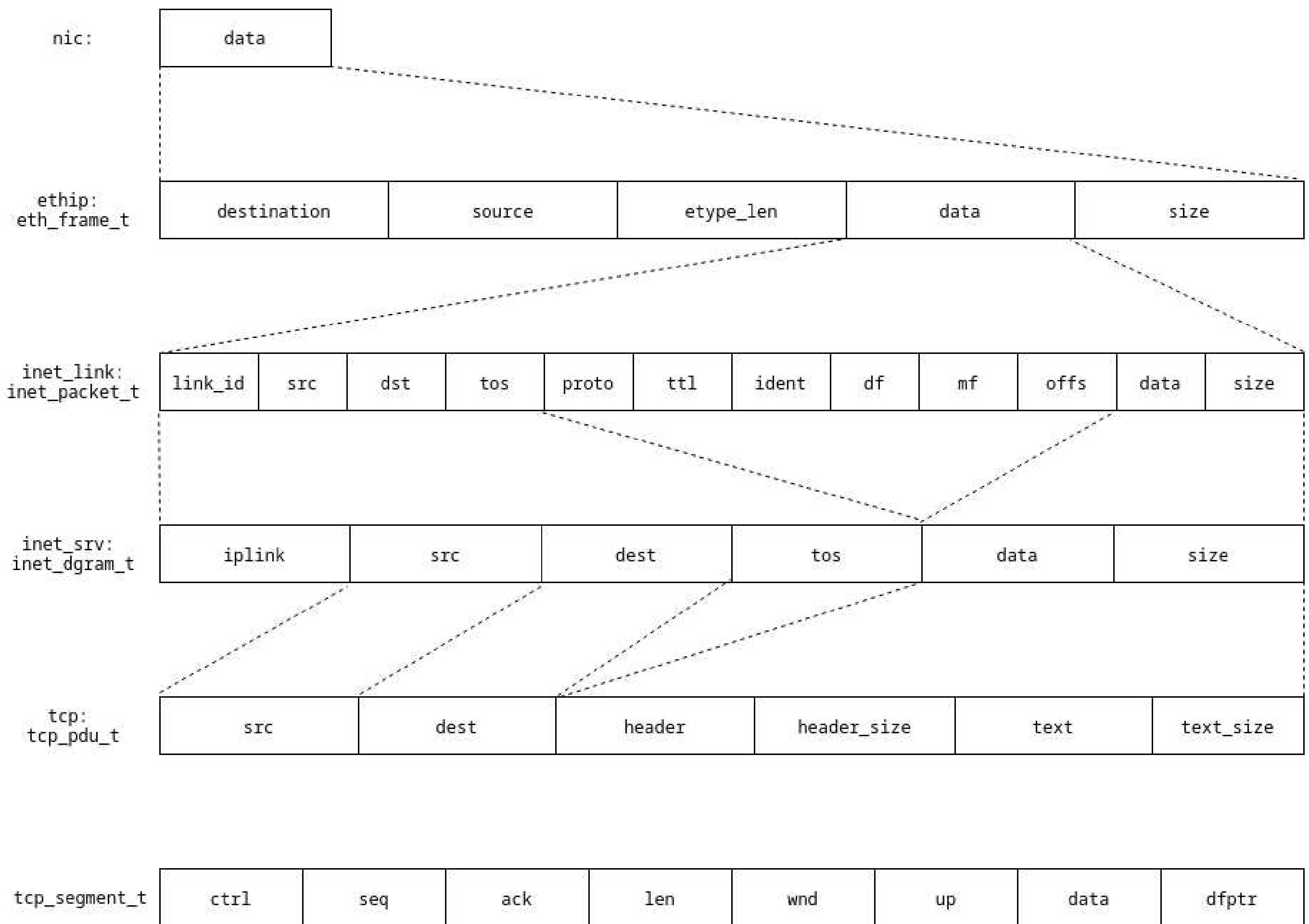


Figure 3.4: Ethernet frame from driver up the TCP stack of HelenOS. On the side there is the name of the service and data type of the structure in HelenOS.

Transport protocols can send outgoing packets. The transport protocol servers send source and destination addresses and payload. `inetsrv` encode IP datagram, looks at the destination address. If the destination address is in the same network as link interface, then the link interface is chosen. Otherwise after consulting routing tables the interface of the best router is chosen. Finally, `inetsrv` sends the datagram to the chosen interface via IPC. (8)

`inetcfg` port is used by administration utilities. `inetping` port is used for partial implementation of ICMP protocol. Detailed description of these ports is not necessary for the purpose of this thesis.

3.3.3 Transport layer: tcp/udp

The `tcp` and `udp` servers implement their respective protocols.

`Udp` server holds list of associations. An association is defined by local IP address, local port, foreign IP address and foreign port.

User application can connect to the `udp` server via IPC. The socket is created and bound to the free port. During binding the local port and local IP address are set. After that application sends the *sendto* command to `udp` server via IPC. If socket is unbound, then the server sets it to the free port, creates packet header and sends it to the `inetsrv` over IPC.

`Tcp` server holds list of connections. Connection is defined by local IP address, local port, foreign IP address, foreign port and listen flag.

User application creates connection by sending call via IPC and binding it. During binding `tcp` server sets the local IP address and local port. When receiving *connect* message `tcp` server checks if the socket is already bound. If it is not, the server binds it to free port. Then the server sets foreign IP address and foreign port and performs the handshake.

On *send* IPC call, `tcp` server checks validity of the connection. If the connection is valid, the the server pushes the data into the dispatch queue to be sent to the other side. On *recv* IPC call the server, after checking the validity of the connection, checks receive-queue. If the receive queue is not empty, wanted amount of data is sent via IPC to the application. Otherwise the application is blocked until some data arrives. The application is awakened and the data are sent to it.

3.4 Architecture designs

After exploring TCP stack in HelenOS in section 3.3 we came to conclusion, that we need to dump packets on the lowest level which is driver level. On the driver level the frames are represented as data buffer, which means that they are not parsed into structures and nothing is cut out as can be seen in figure 3.4.

Another reason is as not all packets are propagated to upper levels.

During the work, three different approaches for implementation of the framework were considered. The approaches differ in the degree of driver participation in packet dumping. The section describes approaches with their advantages and disadvantages.

3.4.1 Forwarding server

The idea is to implement a forwarding server. The server would listen to traffic on specific driver and copy and forward packets when requested. On one side, the server would communicate with the driver. On the other side, server would communicate with user application that would take care of packets.

The application would send an IPC request to the server. The server would start listen for traffic and copy packets from `nic` and forward them to the application for dumping. Server would be minimal and all the logic, i.e. filtering, would be in user application. Fig. 3.5 demonstrates the idea. The driver is encapsulated in `nic` structure, therefore on the figure we do not distinguish between different drivers.

Advantages:

- One main entity that would redirect and control communication.
- The server is simple. The logic is in dumping application, therefore application can be replaced without changing the server.
- In case of implementing filtering as a future work, the server would stay without changes. We would change the application or introduce a new application.

Disadvantages:

- The packets would be copied several times between the servers.
- Implementing a new server means that we would need to take care of a new entity.

3.4.2 Dumping Server

The idea is to implement a server that would dump packets. In that approach, all the logic is in the server. On one side, server communicates with the application, receiving requests to start or stop dumping from it. On the other side the server communicates with the driver, listening to traffic and writing packets to file. The approach is illustrated in Fig. 3.6.

Advantages:

- Neither the driver nor the application interferes with the file system. The server takes care of writing to the file.

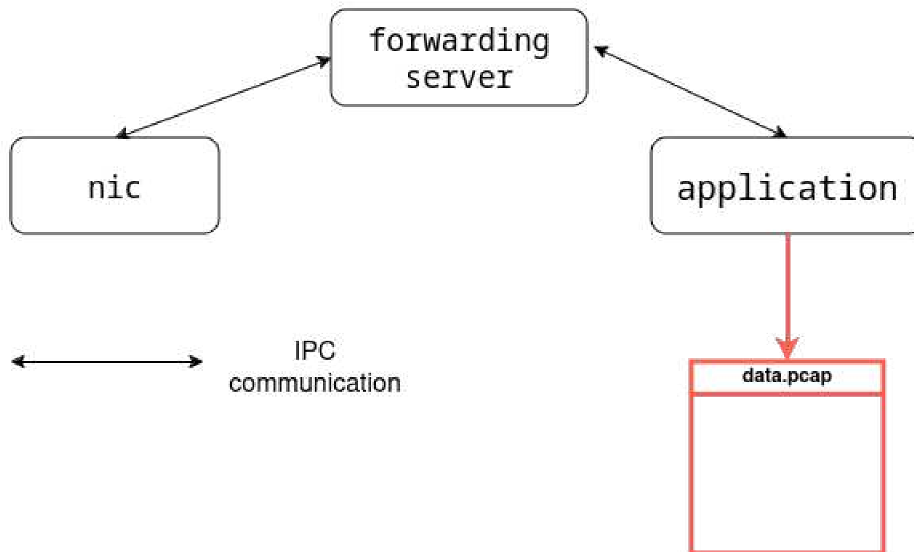


Figure 3.5: Server as forwarding and controlling unit

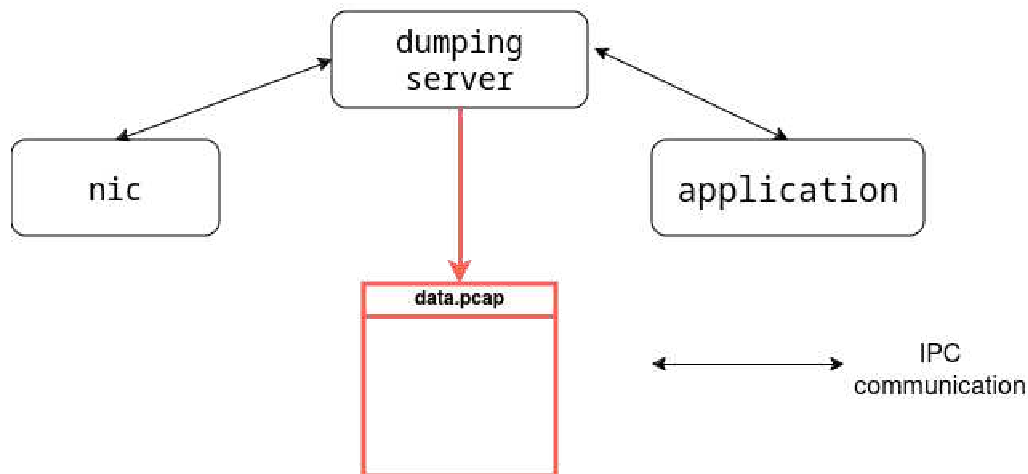


Figure 3.6: Server as dumping unit

- The server is complex, as it holds all the logic, but the application is simple.
- We reduce copying of the packets via the IPC.

Disadvantages:

- As in previous approach implementing new server is complex and time consuming.
- In case of implementing filtering as a future work, we would need to modify server code significantly.

3.4.3 Application and driver

The idea is to implement a direct interaction between the CLI control application and the driver. In that approach, we do not have server, therefore

responsibilities are divided between the driver and the application.

The driver is responsible for dumping the packets, for example writing them to file. The application serves as a control unit that allows user to choose the driver for dumping and send commands for the driver. The approach is illustrated in Fig. 3.7.

Advantages:

- We have reduced copying packets between servers, as driver dumps the packets itself.
- No need to implement new server.

Disadvantages:

- The driver interferes with the file system as the driver itself manages writing to the file.
- In case of implementing filtering as a future work we would need to add non-trivial changes to the application.

3.4.4 Final decision

The first approach 3.4.1 was dismissed because we concluded that the server does not store any state. Its purpose would only be forwarding. For that purpose, we can use other built-in components in HelenOS.

The second approach 3.4.2 was partially dismissed for the same reason as the first, we did not want to implement a new server. Another reason is redundant copying of the data that we wanted to avoid.

We chose the third approach 3.4.3. The approach was chosen because implementing the interaction between the application and the driver is not as complex as the alternatives. We do not need to implement a new server, because for the binding driver and application we will use the existing `locsrv` location service.

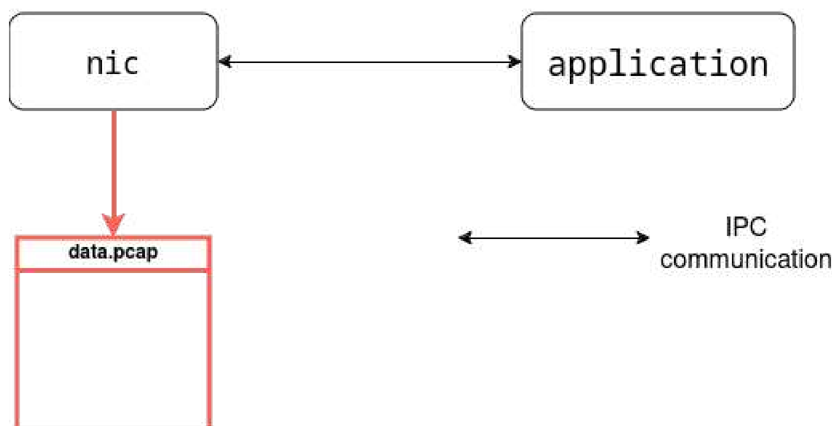


Figure 3.7: Direct interaction of the driver and the application as an approach

We will not have redundant copying of the packets with this approach. As far as code maintenance is concerned, the latter approach is the most attractive since we have fewer entities to implement and therefore fewer components to maintain.

The disadvantages of the chosen approach are that the driver interferes with the file system as the driver itself writes packets to the file, or, in general, the driver executes the writing. Another downside is that in case of extensions, like filtering, support for other formats to store data, we would need to add changes to the control application code. But we can reconcile with that because the advantages of the approach are worth it.

During analysis of the drivers in HelenOS we found out that for their identification long names in format "\hw\sys\00:0x.x\..." are used. We do not want to complicate user experience, therefore we will implement support for indices in control application for choosing the driver for dumping.

Apart from the control CLI application that would manage dumping, we will implement a small CLI "pcapcat" application that will parse the PCAP file and print out to standard output basic information of the packets. The application would be useful for debugging purposes in context of not only our work but future work with the HelenOS TCP/IP stack.

3.5 Ideas for testing

As mentioned in Section 2.6 HelenOS has CI sub-project. We will use it and develop several scenarios with commands and expected results. That would test the functionality of the framework. Some tests require starting HelenOS with two network drivers. CI project has minimal functionality and did not provide opportunity to specify how many or what network drivers should be chosen when starting the system. We extended CI project code by adding new option "–two_network_drvs" with help of which user can start HelenOS with two network drivers. From the functionality point of view we do not need to test all drivers. Therefore option does not offer a choice of drivers, but starts the system with e1k and ne2k drivers. Details about extension of the CI project and test scenarios are in section 5.1.1.

As another test we will run in parallel dumping on the HelenOS and on the host to test that all the data are captured. Each packet record also has a timestamp. Therefore comparing results becomes complicated as files can not be compared one-to-one. Detailed description of the conducting of the test and results is in the section 5.1.2.

Chapter 4

Implementation

In this chapter we describe the process of the implementing framework to support dumping packets. We will dive into the API that HelenOS provides us with. We implemented a library `pcap`, an application `pcapctl`, and made some changes in the network drivers code.

The library consists of several modules that serve different purposes. The modules are: work with PCAP format, interface for the driver, server and client sides of the IPC communication, and interface for the application `pcapctl`. The framework uses service `locsrv`. The figure 4.2 demonstrates key components that take part in the dumping.

During work we implemented support for PCAP file format in HelenOS. Subsection 4.1.1 describes the result. We created a dumper structure that provides functions to dump and destination buffer for dumping. Subsection 4.1.3 describes the structure. The dumper is a part of network drivers. We modified the network drivers code. Subsection 4.1.2 describes the result. We created a service that was able to find and communicate with the driver, as, based on section 3.4.4, the driver is responsible for dumping packets. Section 4.1.5 describes the process. For the user, we created an application that can start and stop the dumping for the chosen driver. Section 4.2 describes the process and the result. The graph 4.1 demonstrates the whole design.

The framework was written in C language, as it is the main language of HelenOS. Appendix F describes how to use the framework to add support for packet dumping to a driver in general.

4.1 pcap library

The library provides support for working with PCAP format, an interface for drivers, and encapsulation of IPC.

4.1.1 PCAP format for dumping

As was described in section 3.1, the chosen format is PCAP. File has header and specific header for every packet as was described in section 3.2. Both headers

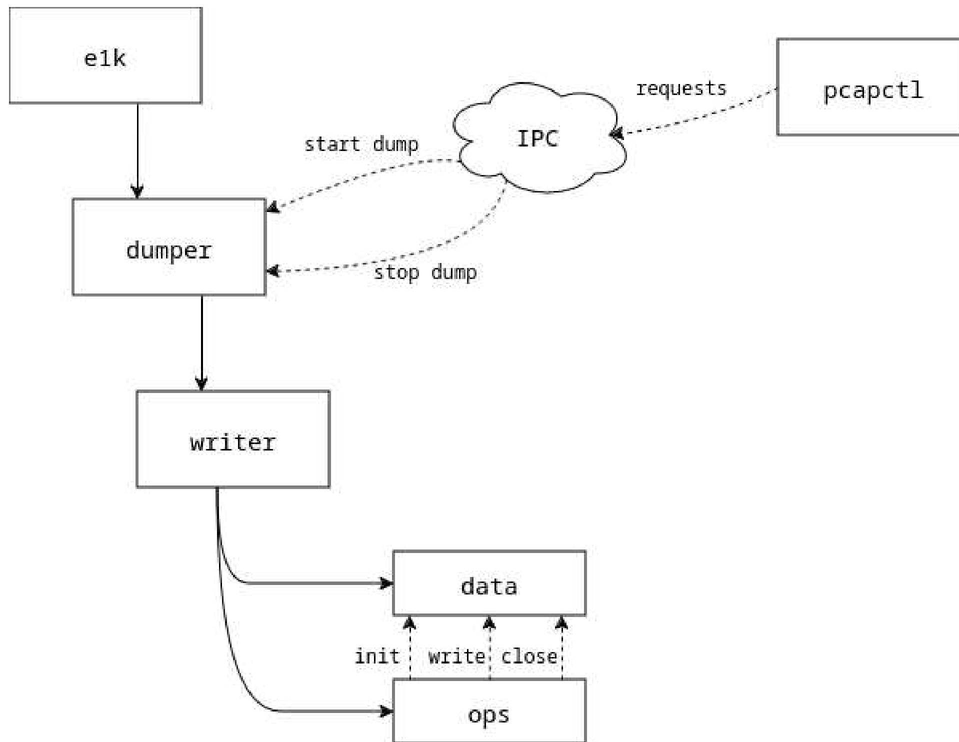


Figure 4.1: Framework design.

```c
typedef struct {
    uint32_t magic_number;
    uint16_t major_v;
    uint16_t minor_v;
    uint32_t reserved1;
    uint32_t reserved2;
    uint32_t snaplen;
    uint32_t additional; // Link-layer header type for packets.
} pcap_file_header_t;
```

Listing 1: PCAP format header.

are represented as `struct` as can be seen in Listing 1 and 2. Utility functions such as setting time and filling structures are in the module as well. The most interesting header field is Link-Layer type, fig. 3.1. The default value is IEEE 802.3 Ethernet. The value does not change for wireless connections, because Wireshark and TCPdump do not distinguish between them. (9)

4.1.2 Modification of the NIC code

As was described in section 3.4.4, the chosen approach is the driver dumping the packets. During our implementation we were able to add support for dumping packets to every network driver. We did not need to modify the code of every network driver in HelenOS. All network drivers are encapsulated by the `nic_t` structure; therefore, we only needed to modify the code of nic library and

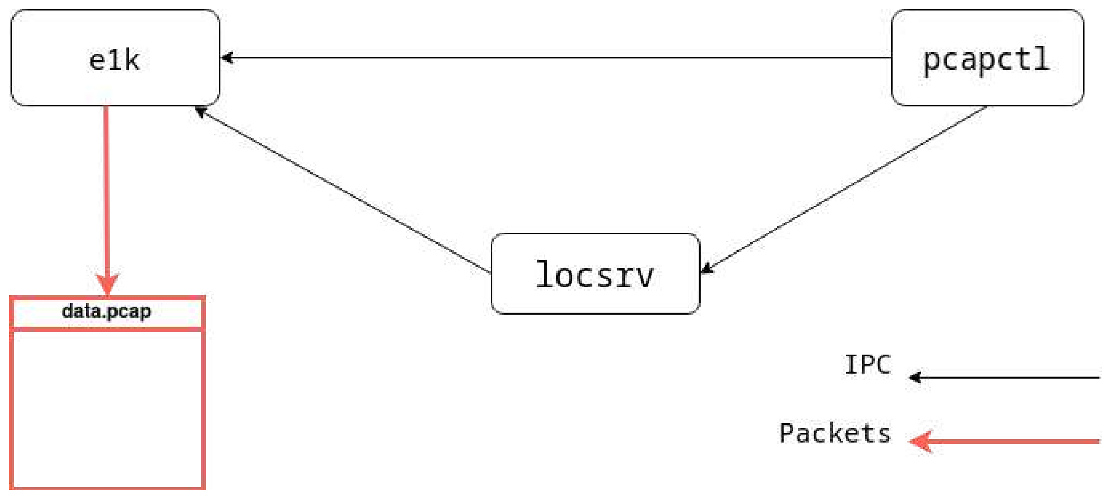


Figure 4.2: Components that participate in the process. As a driver, example e1k is demonstrated. locsrv serves as a broker during the initialization of the dumping; after that, it does not participate in any communication. Described in section 4.1.4.

```
typedef struct pcap_packet_header {
    uint32_t seconds_stamp;
    uint32_t magic_stamp;
    uint32_t captured_length;
    uint32_t original_length;
} pcap_packet_header_t;
```

Listing 2: Packet header.

find the right place with the whole frame to dump instead of changing the code of every driver. Because of such modification, any network driver that will be implemented in the future will automatically have support for packet dumping. Changes in the `nic` library were not big as well. In listings 5 and 4, added lines of code are highlighted.

We created a network driver interface for our framework and added a new field to the generic network driver structure.

Structure `nic_t` encapsulates every network driver, and instead of modifying the code of every driver, we decided to make changes in library `nic`. That way we did not need to interfere with most of the drivers code, and if a new network driver is implemented in the future in HelenOS, it will have the functionality for dumping packets automatically.

The interface we created consists of two functions as seen in Listing 3. Function `pcapdump_init` is used during the initialization of the network driver. The function initializes dumper in the network driver and creates a port for IPC communication. IPC communication is described in more detail in section 4.1.4. Function `pcapdump_packet` is used in two places in code: after receiving a packet and before sending one. The function dumps the packet.

```
extern errno_t pcapdump_init(pcap_dumper_t *);
extern void pcapdump_packet(pcap_dumper_t *, const void *, size_t);
```

Listing 3: Driver interface for communication with our framework.

```
...
#include <pcapdump_drv_iface.h>
...
errno_t nic_send_frame_impl(ddf_fun_t *fun, void *data, size_t size)
{
    nic_t *nic_data = nic_get_from_ddf_fun(fun);

    fibril_rwlock_read_lock(&nic_data->main_lock);
    if (nic_data->state != NIC_STATE_ACTIVE || nic_data->tx_busy) {
        fibril_rwlock_read_unlock(&nic_data->main_lock);
        return EBUSY;
    }
    pcapdump_packet(nic_get_pcap_dumper(nic_data), data, size);
    nic_data->send_frame(nic_data, data, size);
    ...
}
...
```

Listing 4: Changes in file nic_impl.c. Highlighted lines were added during work.

Listings 4 and 5 demonstrate changes in nic library code. The dumper and its part in the framework are described in section 4.1.3.

4.1.3 Packet dumper

We worked on the framework with the thought that there will be extensions of the functionality in the future. Therefore we designed the framework to be extensible. There are several options for how to do that. One of such options is the `ops` structure. For our framework we chose that option because it is already used in other parts of HelenOS, and it is relatively easy to change behavior, for filtering for example, during runtime by interchanging `ops` structures. With `ops` structures we simulate interfaces from object-oriented languages.

That is why two structures, `pcap_writer_t` and `pcap_writer_ops_t`, were implemented. With changing writer and writer operations, a programmer can implement more sophisticated filtering in the future without changing key parts of the framework. Listing 6 demonstrates assignment of the chosen `ops` structure to the dumper during the initialization phase of dumping. Below are details of all three structures.

Structure `pcap_dumper_t` is responsible for dumping data. The structure is in Listing 7. The dumper structure has a mutex field for synchronization, the flag `to_dump`, which indicates whether the frame should be ignored by the dumper or dumped to the destination buffer, and `writer` structure that is responsible for

```c
...
#include <pcapdump_drv_iface.h>
...
void nic_received_frame(nic_t *nic_data, nic_frame_t *frame)
{
    ...
    pcapdump_packet(nic_get_pcap_dumper(nic_data),
        frame->data, frame->size);
    fibril_rwlock_read_lock(&nic_data->rxc_lock);
    nic_frame_type_t frame_type;
    bool check = nic_rxc_check(&nic_data->rx_control, frame->data,
        frame->size, &frame_type);
    fibril_rwlock_read_unlock(&nic_data->rxc_lock);
    ...
}
...

nic_t *nic_create_and_bind(ddf_dev_t *device)
{
    nic_t *nic_data = nic_create(device);
    if (!nic_data)
        return NULL;

    nic_data->dev = device;

    errno_t pcap_rc =
        pcapdump_init(nic_get_pcap_dumper(nic_data));
    if (pcap_rc != EOK) {
        printf("Failed creating pcapdump port\n");
    }

    return nic_data;
}

...

pcap_dumper_t *nic_get_pcap_dumper(nic_t *nic_data)
{
    return &nic_data->dumper;
}
...
```

Listing 5: Changes in file nic_driver.c. Highlighted lines were added during work.

```
static pcap_writer_ops_t ops[4] = { file_ops, short_file_ops,
append_file_ops, usb_file_ops };
...
errno_t pcap_dumper_set_ops(pcap_dumper_t *dumper, int index)
{
    fibril_mutex_lock(&dumper->mutex);
    dumper->writer.ops = &ops[index];
    fibril_mutex_unlock(&dumper->mutex);
    return EOK;
}
```

Listing 6: Assigning of chosen ops structure before starting of the dumping.

```
typedef struct pcap_dumper {
    fibril_mutex_t mutex;
    bool to_dump;
    pcap_writer_t writer;
} pcap_dumper_t;
```

Listing 7: Dumper structure.

working with the destination buffer. Writer structure is in Listing 8. Writer has two key parts: destination dumping buffer and writer operations, which are functions for initializing, deinitializing buffer, and writing to it. Writer operations structure pcap_writer_ops_t is in Listing 9.

In our implementation we distinguish between writer initialization and dumper initialization. Writer initialization is initialization of the dump operations, which is structure pcap_writer_ops_t, and initialization of the destination buffer data in pcap_writer_t structure. Dumper initialization is initialization of the pcap_dumper_t structure. Dumper initialization takes place during initialization of the driver, and writer initialization takes place during the start of the dumping process.

By distinguishing these steps, we introduce the concept of basic filtering with the help of different pcap_writer_ops_t structures. The programmer defines different writer operations, and the user can decide during runtime what operations to use for dumping.

The benefit of such solution is that afer implementing new instance of pcap_writer_t or pcap_writer_ops_t the only thing that must be done is recompiling, no changes

```
struct pcap_writer {
    void *data;
    pcap_writer_ops_t *ops;
};
```

Listing 8: Writer structure.

```c
typedef struct {
    errno_t (*open)(pcap_writer_t *, const char *);
    size_t (*write_u32)(pcap_writer_t *, uint32_t);
    size_t (*write_u16)(pcap_writer_t *, uint16_t);
    size_t (*write_buffer)(pcap_writer_t *, const void *, size_t);
    void (*close)(pcap_writer_t *);
} pcap_writer_ops_t;
```

Listing 9: `pcap_writer_ops_t` structure.

of the existing code are needed.

As an example, our implementation has four types of writer operations:

- Network packets dumping to file.
- Shorter version of network dumping to file, first 60 bytes of each packet.
- Network packets dumping to existing file. Instead of creating a new file and writing the file header to it, this version opens the existing file and appends packets to the end of the file.
- USB packets dumping to a file.

Operations can be seen in Listing 10. Writer operations allow us to hide internal details from the dumper, and thus changing the destination buffer and/or writer operations does not change the code of the whole framework. Choosing writer operations during runtime is described in section 4.2.

One of the `ops` structures is `usb_file_ops` for dumping usb packets. Dumping of USB packets is only in the beginning and is not fully implemented. There are several drivers in HelenOS that work with USB, for which USB dumping can be implemented in the future. For now, the `usb_file_ops` structure is only used for debugging purposes.

Working with `pcap_dumper_t` is done via functions in Listing 11. The functions are called on the server side of the IPC communication. Functions set and unset `to_dump` flag, set writing operations, and dump packets.

4.1.4 Discoverability of the driver

HelenOS has a `locsrv` service that stores "absolute paths" of the devices and has categories, which are groups where `locsrv` can add the driver.

For our framework we created the category "pcap". We add devices that can dump packets to the category "pcap". During initialization of the system, network driver (or drivers, if several) is added to the category "pcap". During the initialization of the driver, the port with the callback connection is created. Then before the start of the dumping process, the driver is found through the category "pcap" and the request to start is redirected via callback connection to the driver.

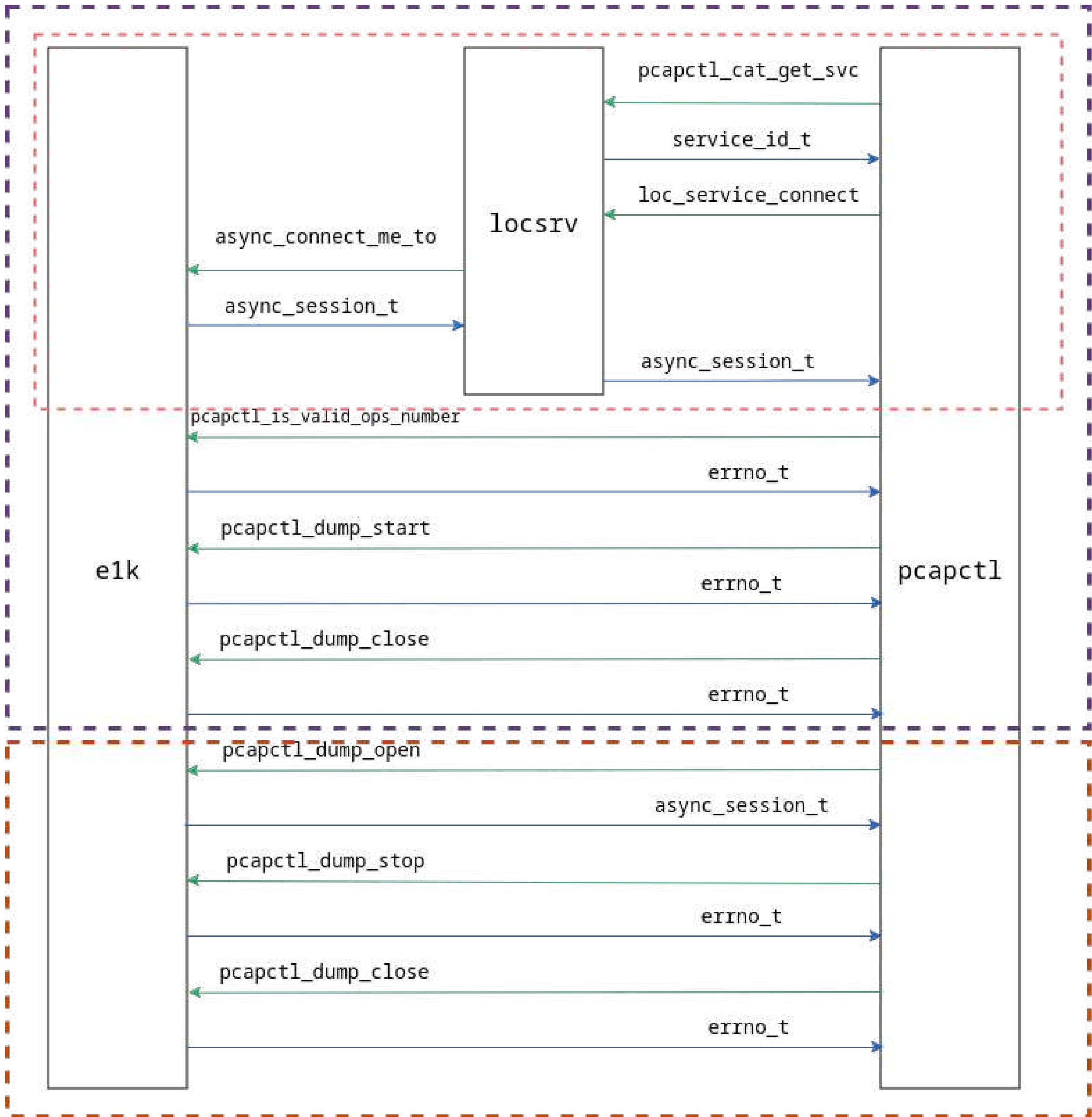


Figure 4.3: IPC between key components, taking part in dumping packets

```
static const pcap_writer_ops_t file_ops = {
    .open = &pcap_writer_to_file_init,
    .write_u32 = &pcap_file_w32,
    .write_u16 = &pcap_file_w16,
    .write_buffer = &pcap_file_wbuffer,
    .close = &pcap_file_close
};

static const pcap_writer_ops_t short_file_ops = {
    .open = &pcap_writer_to_file_init,
    .write_u32 = &pcap_file_w32,
    .write_u16 = &pcap_file_w16,
    .write_buffer = &pcap_short_file_wbuffer,
    .close = &pcap_file_close

};

static const pcap_writer_ops_t append_file_ops = {
    .open = &pcap_writer_to_file_init_append,
    .write_u32 = &pcap_file_w32,
    .write_u16 = &pcap_file_w16,
    .write_buffer = &pcap_file_wbuffer,
    .close = &pcap_file_close
};

static const pcap_writer_ops_t usb_file_ops = {
    .open = &pcap_writer_to_file_usb_init,
    .write_u32 = &pcap_file_w32,
    .write_u16 = &pcap_file_w16,
    .write_buffer = &pcap_file_wbuffer,
    .close = &pcap_file_close
};

static pcap_writer_ops_t ops[4] =
    {file_ops, short_file_ops, append_file_ops, usb_file_ops};
```

Listing 10: Different writer operations .

```
extern errno_t pcap_dumper_start(pcap_dumper_t *, const char *);
extern void pcap_dumper_stop(pcap_dumper_t *);
extern int pcap_dumper_get_ops_number(void);
extern errno_t pcap_dumper_set_ops(pcap_dumper_t *, int);
extern void pcap_dumper_add_packet(
    pcap_dumper_t *, const void *data, size_t size);
```

Listing 11: Function to work with `pcap_dumper_t`.

```
extern errno_t pcapctl_dump_open(int *, pcapctl_sess_t **);
extern errno_t pcapctl_dump_close(pcapctl_sess_t *);

extern errno_t pcapctl_dump_start(const char *, int *, pcapctl_sess_t *);
extern errno_t pcapctl_dump_stop(pcapctl_sess_t *);

extern errno_t pcapctl_list(void);

extern errno_t pcapctl_is_valid_device(int *);
extern errno_t pcapctl_is_valid_ops_number(int *, pcapctl_sess_t*);
```

Listing 12: Client side functions.

```
extern void pcapdump_conn(ipc_call_t *, void *);
```

Listing 13: Server side callback connection function.

4.1.5 Dump controlling

The IPC and asynchronous communication were described in section 2.1. From the client's point of view, there are request functions. Request functions include starting and stopping dumping packets.

The client side contains functions in Listing 12. The first two functions are responsible for starting and ending asynchronous session `async_sess_t` for sending messages via `locsrv`. Two second functions are responsible for sending requests to the driver that dumps packets. Requests are `PCAP_CONTROL_SET_START` to start dumping and `PCAP_CONTROL_SET_STOP` to stop it. The fifth function gets the list of devices that can dump packets; they are in the category "pcap" and the last two functions check whether the index of chosen operations and devices is an index of valid operations and driver.

Figure 4.3 demonstrates communication between application `pcapctl`, `locsrv`, and driver (in our example, e1k). The figure shows sending a request to start dumping inside the purple dotted frame and a request to stop dumping inside the red dotted frame. Inside the pink dotted frame, the function `pcapctl_dump_open` is demonstrated in more detail.

The server side contains a callback connection function as shown in Listing 13. As was described in section 4.1.4, the device has a port on which it listens for requests. During initialization of the device, the port is created with the callback function from Listing 13. The function redirects the request based on the type of the sent requests. The framework has three types of requests shown in Listing 14.

Redirected requests are processed with the help of dumper functions described in section 4.1.3 and shown in Listing 11.

```c
typedef enum {
    PCAP_CONTROL_SET_START = IPC_FIRST_USER_METHOD,
    PCAP_CONTROL_SET_STOP,
    PCAP_CONTROL_GET_OPS_NUM,
} pcap_request_t;
```

Listing 14: IPC requests for the framework.

```
$ pcapctl -N data.pcap
Start dumping on device - 0, ops - 0.
$ ping 8.8.8.8
...
$ pcapctl -t
Stop dumping on device - 0.
```

Listing 15: Simple ping 8.8.8.8 to file.

4.2 Applications

During work, two applications were implemented. `pcapctl` is responsible for managing dumping on drivers. `pcapcat` prints contents of PCAP file to standard output.

4.2.1 `pcapctl`

User can use the framework via the designed console application `pcapctl`. Apart from standard requests like start and stop, user can specify which kind of operations to use for dumping, which are described in section 4.1.3, and choose what device will dump the packets. An example of the usage is in Listing 15. As we mentioned before and as can be seen from the Listing 15 for the identification of the device simple indices are used. For listing full names of the devices and their indices user can use "`pcapctl -l`". After stopping dumping, the user can get the file outside HelenOS and open it with Wireshark to see dumped packets or use utility `pcapcat` inside HelenOS to print basic information about packets from file. Full user documentation with more examples and a guide on how to get the file outside HelenOS is in appendix B.

The application uses wrapper functions that call functions responsible for the client side of the IPC communication described in subsection 4.1.5. As a parameter, functions pass the current session `pcapctl_sess_t`. The session is used for the IPC communication as described in section 2.1.

4.2.2 `pcapcat`

The application was implemented for debugging purposes mainly. It allows printing some basic information about dumped packets to standard output in HelenOS without the necessity to get a file outside HelenOS to see its contents.

```
$ pcapcat d.pcap
LinkType: 1
Magic number:  0xa1b2c3d4
0001) [IPv4] IP header: 20B, payload: 556B, protocol: 0x11,
Source IP: 10.0.2.2, Destination IP: 255.255.255.255
0002) [IPv4] IP header: 20B, payload: 264B, protocol: 0x11,
Source IP: 0.0.0.0, Destination IP: 255.255.255.255
0003) [ARP] Sender MAC: 52:54:00:12:34:56, Sender IP: 10.0.2.15
Target MAC: ff:ff:ff:ff:ff:ff, Target IP: 10.0.2.2
...
```

Listing 16: Example of printing out dumped traffic to standard output.

```
typedef struct {
    uint32_t linktype;
    void (*parse_file_header)(pcap_file_header_t *);
    void (*parse_packets)(FILE *, int, bool);
} linktype_parser_t;
```

Listing 17: Parser structure for specific linktypes.

Listing 16 shows an example of printing out the contents of file d.pcap to standard output. Appendix C demonstrates usage in more detail.

At the moment, the application supports parsing of `LINKTYPE_ETHERNET` types of files. However, the application is implemented with the thought of future extensibility, and implementing parsing of other link-layer types can be easily added to the application.

Parsers for different link-layer types must be implemented with the help of the structure in Listing 17.

The application has an array of currently implemented parsers, as Listing 18 shows, and when a file is given for parsing, the application finds link-layer type in the file header and tries to find parser structure with the same link-layer type value. If successfully found, then the file is parsed; otherwise application notifies user that the parser for specific link-layer type is not implemented.

```
static const linktype_parser_t parsers[1] = {eth_parser};
```

Listing 18: Array of currently implemented parsers for `pcapcat` application.

Chapter 5

Benchmarking and testing

After creating the framework, we tested it in different ways. We did not run HelenOS on real hardware, as it was problematic to find any. But we covered functionality with tests described below and tested it in QEMU.

5.1 Testing

We added several automated tests to test most kinds of behaviors. Apart from that, we verified that our framework captures all traffic.

5.1.1 Automated test scripts

As was mentioned in Section 3.5 we decided to test the functionality of our framework using the HelenOS CI subproject. CI project serves two purposes: automated builds and pre-merge tests. We are interested in testing.

HelenOS does not have proper implementation of shell that would support conditions, therefore we cannot write tests inside the system and need to use CI subproject. CI project runs QEMU and with help of YML scripts, that have commands and expected output, simulates user input and reads output from QEMU to compare it with expected. If the expected output and actual are different, then scenario execution stops, and the test is considered failed. Otherwise, the test is successful.

CI project has minimal functionality, and we extended it by adding new option `--two_network_drvs` with help of which user can start HelenOS with two network drivers: e1k and ne2k. Some of our tests need two network drivers.
Scenarios for our framework are in the folder scenarios/pcapctl/ of the CI project.

The scenarios we designed are as follows:

- `pcapctl_default.yml` - starting dumping on default device (e1000) with default writer operations (dumping full packet to the file), default ping (8.8.8.8) and stopping the dumping. The result file must be of a definite size and have a request and answer packet for ping.

- `pcapctl_append_writer_ops.yml` - starting dumping with default writer operations to the file, stopping dumping, and starting again with "append" writer operations to the same must open the file and without adding header file and overwriting file and start dumping packets at the end of the file.
- `pcapctl_force_flag.yml` - trying to open an existing file for dumping should not be successful without using flag `--force`.
- `pcapctl_list.yml` - listing devices that can dump packets. The test is useful when running HelenOS with two network drivers. Appendix E describes how to start HelenOS with two network drivers in context of CI tests.
- `pcapctl_short_ops.yml` - running commands with short options. The test must have the same results as `pcapctl_default.yml`.
- `pcapctl_short_writer.ops` - dumping packets to the file using short writer operations. After stopping, the file must be of a certain size.
- `pcapctl_two_drvs.yml` - dumps different traffic on different network drivers to two different files. After stopping, the files must be of a certain size.
- `pcapctl_two_files_ok.yml` - starting dumping on default device to first file, stopping dumping on default device, starting dumping on default device to second file, first file must not change size. After stopping, the files must be of a certain size.
- `pcapctl_two_files.yml` - starting dumping to file and without stopping trying to start dumping to second file. The second starting must be unsuccessful, and dumping to the first file must not be impacted.
- `pcapctl_user_friendly.yml` - testing user-friendly options to start dumping with basic writer operations.
- `pcapctl_user_friendly_short.yml` - using short variants of options instead of long in previous test.

Unfortunately, even such testing is not ideal, as we can only assert that the utility behaves the way it should and assert the files are of certain size, but we cannot check the content of the files.

Appendix E describes how to run tests for HelenOS locally. (2)

5.1.2 Host network

We conducted a test to confirm that our framework dumps all packets that pass through the network driver.

We configured a bridge on the host machine to which we connected HelenOS. The bridge configuration is shown in Fig. 5.1. QEMU was connected to the host network. The command with which we started HelenOS is in the listing 19. HelenOS was tunneled to the host network.

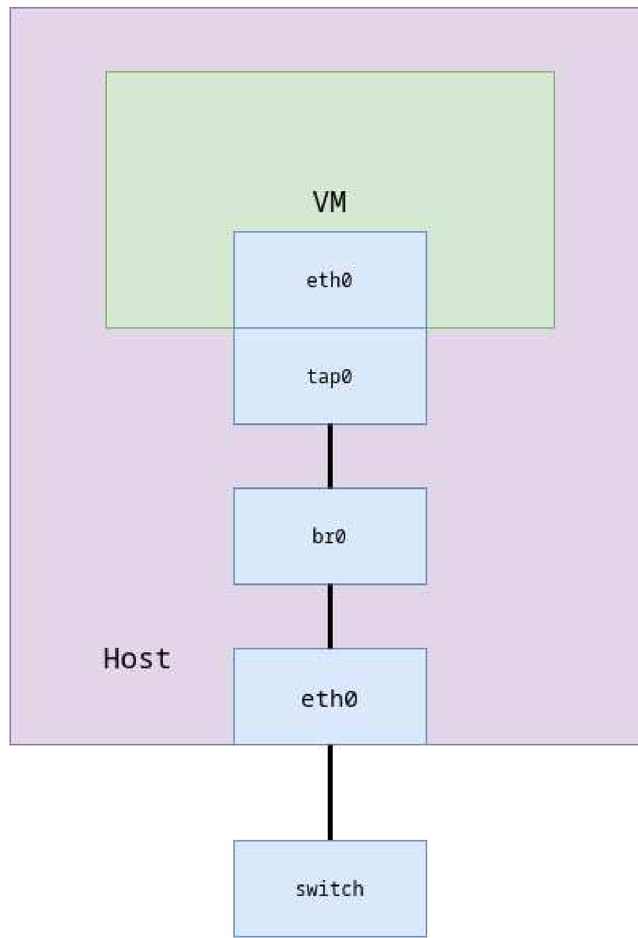


Figure 5.1: Bridge configuration.

In HelenOS we used our command line utility `pcapctl` to dump packets; on host we used `tcpdump` to dump traffic on tap0. We ran a web server inside HelenOS to create traffic on tap0. After stopping dumping and getting the files outside, we compared them in Wireshark.

After applying filter from Listing 20, on file from host machine, the contents of the files were the same. The use of this filter is necessary when comparing files because QEMU does not allow STP and ICMPv6 packets in, and therefore the packets are not in the HelenOS PCAP file. This is a specific behavior of QEMU, to be sure we ran Linux in QEMU with the same configuration, and the resulting Linux PCAP file still did not contain STP and ICMPv6 packets. Files

```
qemu-system-x86_64
-m 256m
-nic bridge,id=n1,mac=52:54:00:BE:EF:01,br=br0
-boot order=d
-cdrom image.iso
-serial stdio
```

Listing 19: QEMU command to start HelenOS with bridge to host network.

```
(!(_ws.col.protocol == "STP")) && !(_ws.col.protocol == "ICMPv6")
```

Listing 20: Wireshark filter.

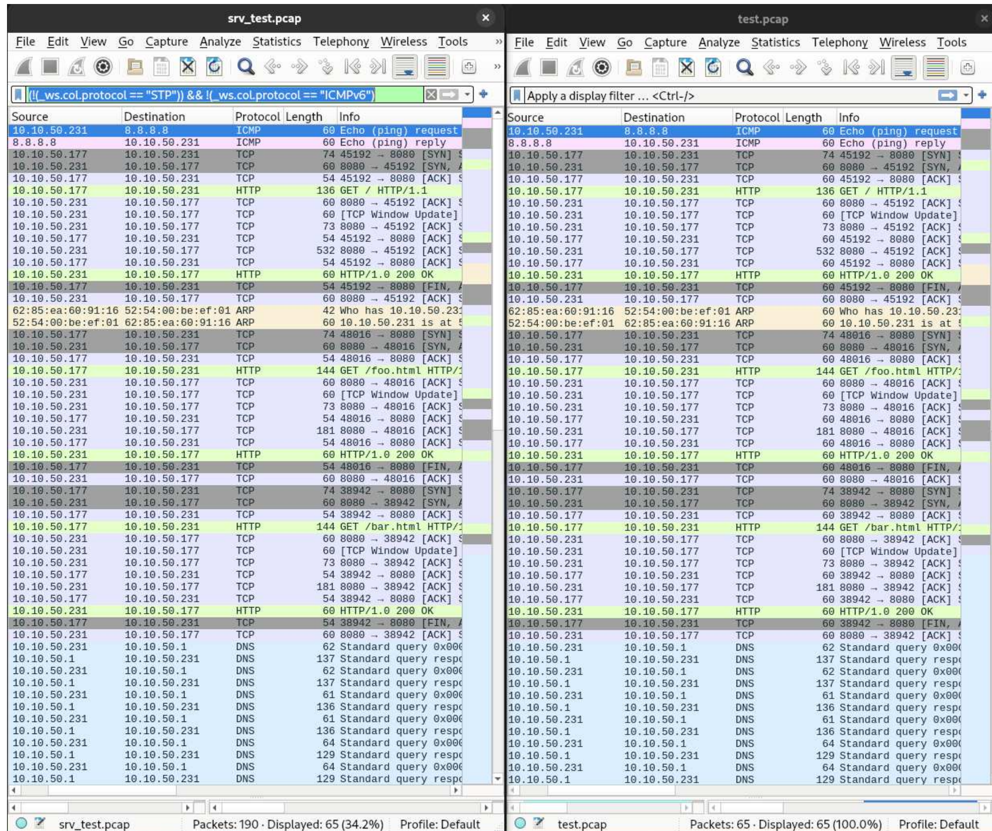


Figure 5.2: Files with dumped traffic from host and HelenOS from left to right.

with applied filter are in fig. 5.2. Files test.pcap and srv_test.pcap from the figure are in the attachments archive.

5.2 Benchmarking

We measured whether dumping of packets has an impact on TCP stack. In HelenOS we started web server and outside we tried to overload HelenOS web server by requests. HelenOS stack has never been subjected to such intensive testing and webserver implementation can not handle the heavy load. Thus, it was necessary to limit the amount of requests, which may negatively affect further observations, because peak performance was not achieved.

We used ab - Apache HTTP server benchmarking tool. HelenOS is run with port forwarding, therefore we were able to run ab utility on the host. We send reading request on `http://localhost:8080/index.html`.

The command we used is in Listing 21. `-n` is the number of requests, `-c` is the level of concurrency, which is how many requests will be sent simultaneously. The maximum number of requests we were able to send is 500, otherwise the

```
ab -n N -c C http://localhost:8080/index.html
```

Listing 21: ab command used for benchmarking.

HelenOS web server would be overloaded and would reset regardless of whether dumping is taking place.

We ran scenario on the mainline of HelenOS, on the branch with support for dumping while dumping packets, and on the branch with support for dumping while no dumping occurred. We tried several parameters; for some parameters, HelenOS web server did not last until the end of the test even on the mainline because of the overload and was reset. As we mentioned earlier webserver can not handle heavy load. Therefore we chose parameters with which the server behaved reasonably and processed all requests. The chosen parameters are 500 for the total number of requests and 4 for the concurrency level. For every of the three cases, the command from the Listing 21 was run 100 times. Figures 5.3, 5.4 and 5.5 demonstrate results.

As illustrated in the figures, the median values remain consistent across all three scenarios, indicating that dumping has no significant impact on the stack behavior under the tested conditions. However, it is important to note that the system did not reach the load capacity 100%, limiting our ability to draw conclusions about behavior under extreme stress. As previously discussed, the current webserver implementation is not equipped to handle high load scenarios. Nevertheless, our tool makes it easier to debug complex parts of the system, like the TCP/IP stack and the webserver in HelenOS, and should help simplify this work in the future.

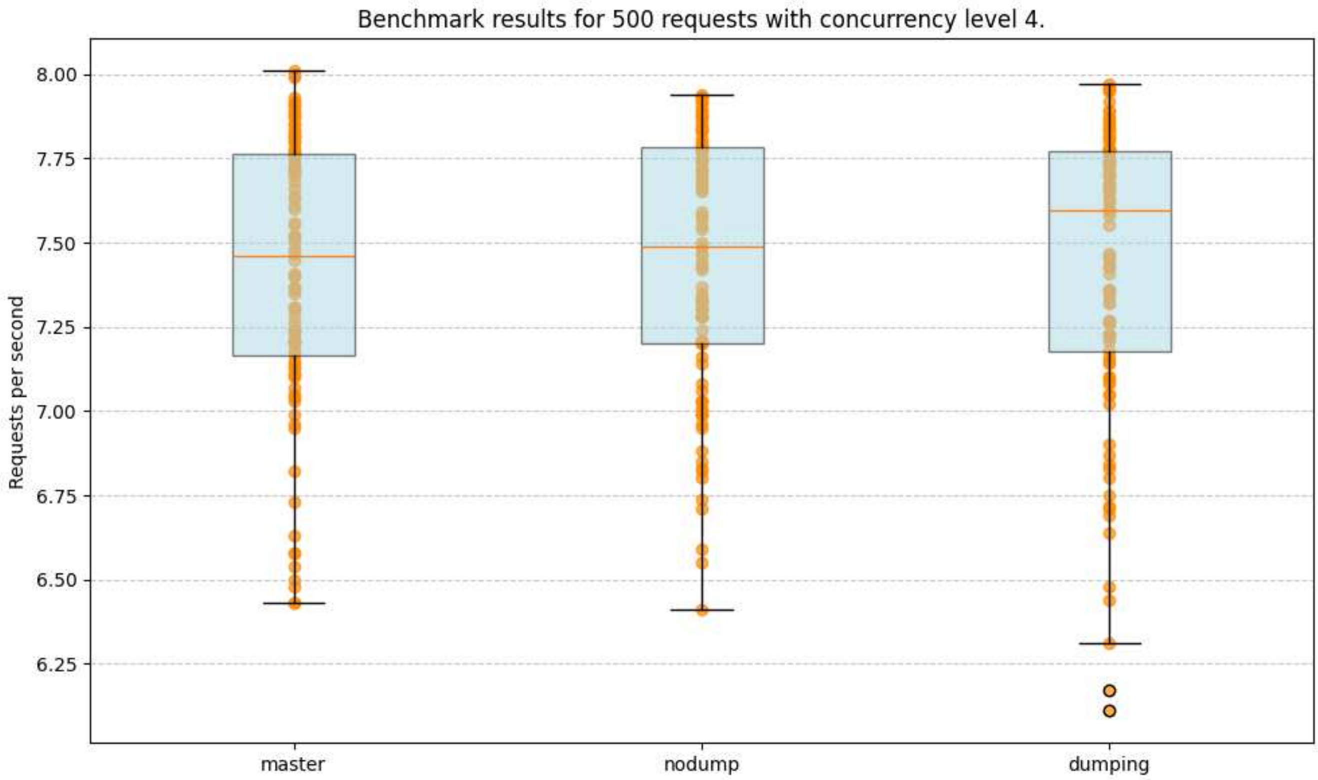


Figure 5.3: Requests per second.

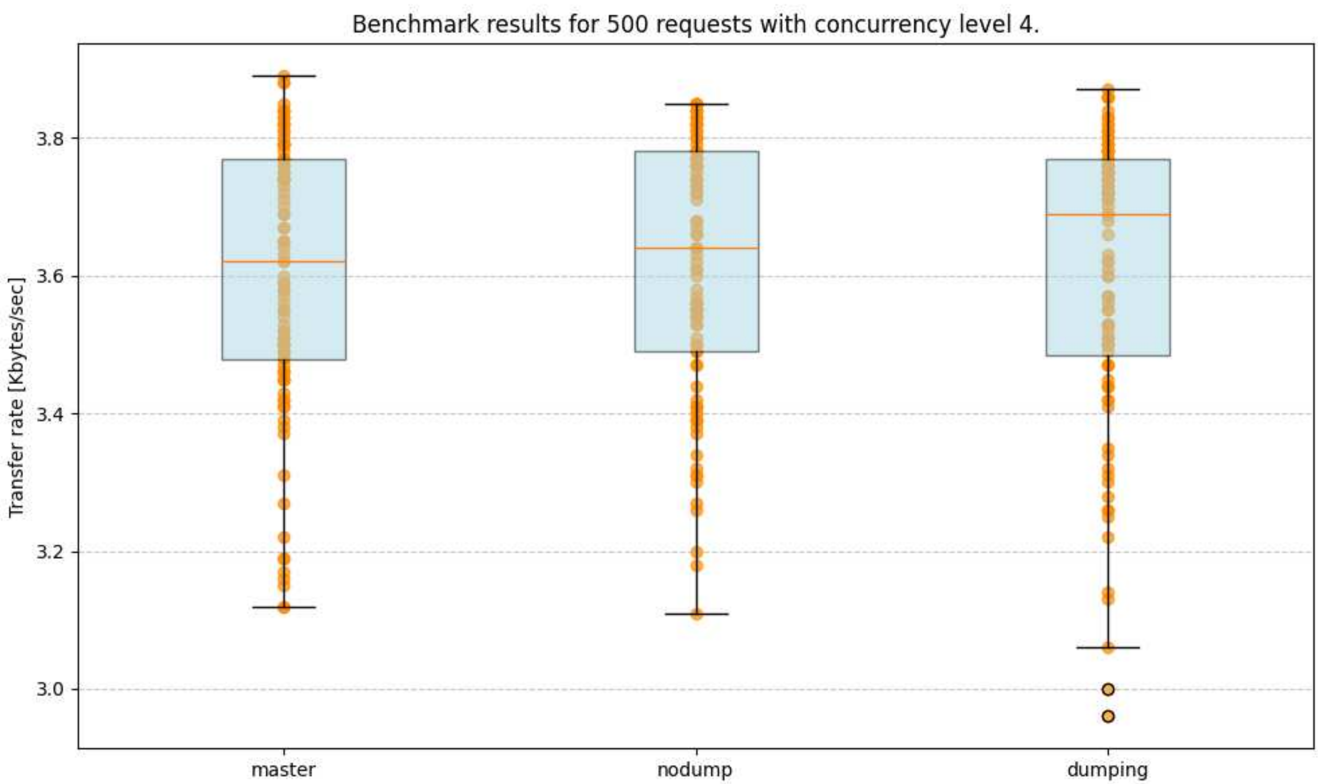


Figure 5.4: Transfer rate through the test.

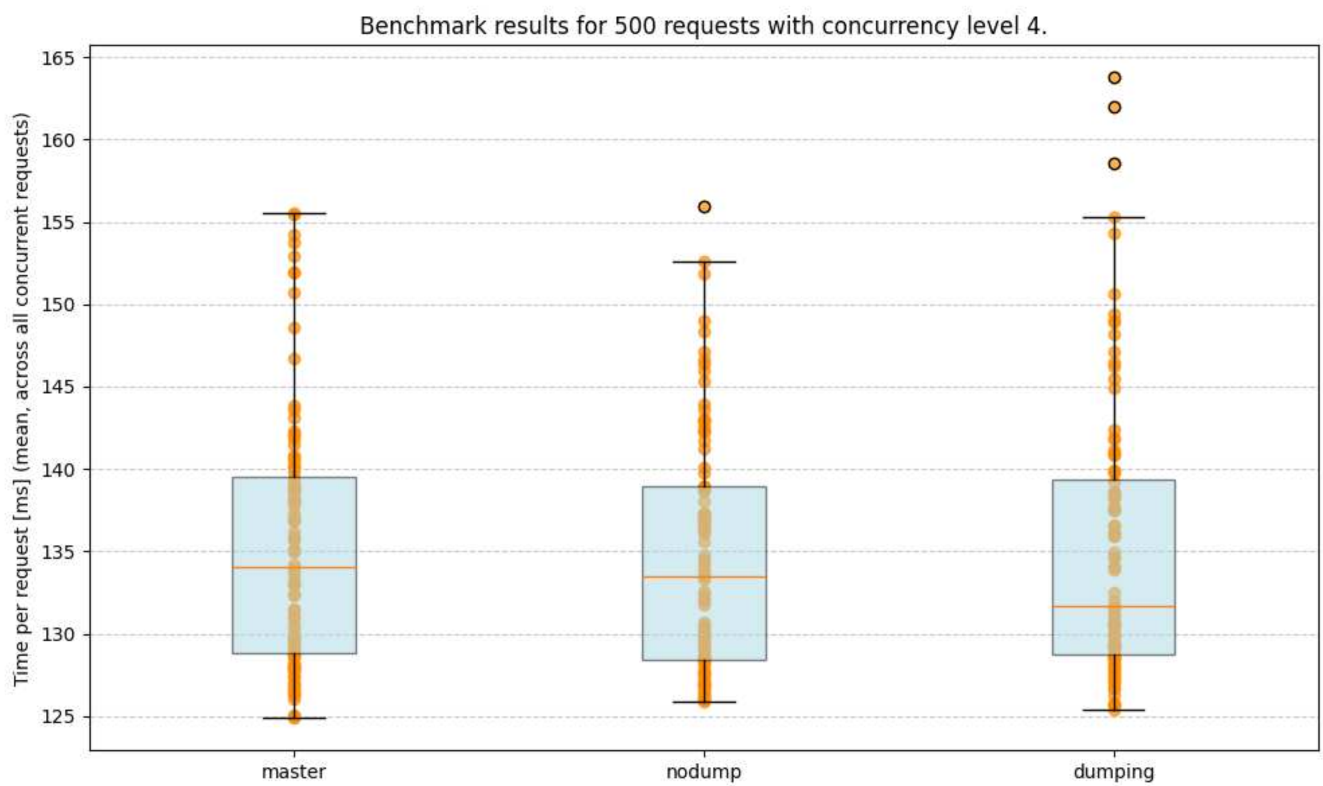


Figure 5.5: Time [ms] per request, mean across all concurrent requests.

Chapter 6

Conclusion

The goals of the thesis were fulfilled. The main goal was to add support for dumping packets on network drivers. After analyzing HelenOS architecture and choosing the right approach, we implemented the framework in a way so that every network driver in HelenOS can dump packets. Moreover, any network driver that will be implemented in the future will automatically be able to dump packets as well without the need to manually add support for dumping packets.

Another goal was to design the framework that will be extensible for further development. The goal was achieved; appendix F describes how to add functionality for dumping packets to any driver. And section 4.1.3 describes how new functionality like different kinds of filtering can be added to the framework without drastic changes.

Lastly, we designed several tests to test the functionality of our framework. The tests are part of sub-project CI. Section 5.1.1 and Appendix E describe the test scenarios and how users can run them locally.

6.1 Future work

The framework is designed in a generic way and can be used to dump packets not only on network drivers. The primary goal of the work was to implement dumping for network drivers, and now among further goals is the extending framework to dump packets on USB drivers.

For this work, we were loosely inspired by `tcpdump`. During the work, only basic filtering and options for command-line utility were implemented. The next step would be to implement more filtering options. Apart from that, `tcpdump` enables dumping some formatted details of the packet to standard output. For now, only contents of PCAP files with linktype `LINKTYPE_ETHERNET` can be parsed and printed to standard output inside HelenOS. The next step in the future would be to extend parsing of more link-layer types.

Appendix F describes how to use the framework to implement dumping on a driver in general.

Bibliography

[1] S. G. Harris, Ed. M. Richardson. PCAP Capture File Format, December 26, 2023. https://ietf-opsawg-wg.github.io/draft-ietf-opsawg-pcap/draft-ietf-opsawg-pcap.html#name-linktype-and-additional-inf, Accessed on December 29, 2023.

[2] HelenOS. GIthub project for CI of HelenOS. https://github.com/HelenOS/ci, Accessed on November 30, 2024.

[3] HelenOS. Github project of HelenOS. https://github.com/HelenOS/helenos, Accessed on November 30, 2024.

[4] HelenOS. HelenOS web, December 16, 2022. http://www.helenos.org/, Accessed on December 09, 2023.

[5] HelenOS. HelenOS/IPC web, February 23, 2023. http://www.helenos.org/wiki/IPC, Accessed on December 09, 2023.

[6] HelenOS. HelenOS/drv, March 01, 2018. http://www.helenos.org/wiki/DeviceDrivers, Accessed on December 09, 2023.

[7] U. L. Richard Sharpe, Ed Warnicke. Wireshark User's Guide Saving Captured Packets. https://www.wireshark.org/docs/wsug_html_chunked/ChIOSaveSection.html, Accessed on December 12, 2023.

[8] M. B. A. Steinhauser. *IPv6 for HelenOS*. phdthesis, Charles University in Prague, Prague, Czech Republic, 2013.

[9] The Tcpdump Group. Link-layer header types. https://www.tcpdump.org/linktypes.html, Accessed on March 20, 2024.

[10] The Tcpdump Group. TCPDump. https://www.tcpdump.org/, Accessed on December 09, 2023.

[11] Wireshark wiki. Development/LibpcapFileFormat. https://wiki.wireshark.org/Development/LibpcapFileFormat, Accessed on April 22, 2024.

List of Figures

3.1 File header. See online reference (1). 9
3.2 Packet record. See online reference (1). 10
3.3 Processing of the `'download http://helenos.org/'` in context of TCP stack in HelenOS 11
3.4 Ethernet frame from driver up the TCP stack of HelenOS. On the side there is the name of the service and data type of the structure in HelenOS. 12
3.5 Server as forwarding and controlling unit 15
3.6 Server as dumping unit 15
3.7 Direct interaction of the driver and the application as an approach 16

4.1 Framework design. 19
4.2 Components that participate in the process. As a driver, example e1k is demonstrated. locsrv serves as a broker during the initialization of the dumping; after that, it does not participate in any communication. Described in section 4.1.4. 20
4.3 IPC between key components, taking part in dumping packets . . 25

5.1 Bridge configuration. 32
5.2 Files with dumped traffic from host and HelenOS from left to right. 33
5.3 Requests per second. 35
5.4 Transfer rate through the test. 35
5.5 Time [ms] per request, mean across all concurrent requests. 36

Appendix A

Starting and running HelenOS

The official GitHub repository of HelenOS describes how to compile HelenOS using toolchain. But if the user does not want to build the system, section A.3 describes how to run HelenOS using attachments in zip. (3)

A.1 Building

Cloning (version with our framework):

```
$ git clone https://github.com/boba-buba/helenos.git
$ cd helenos
$ git checkout topic/packet-capture-rb
```

After cloning we need to do is to install dependencies which can be different for different systems. For Ubuntu 16.04:

```
$ sudo apt install build-essential wget texinfo flex bison dialog
python-yaml genisoimage
```

For CentOS/Fedora:

```
$ sudo dnf group install 'Development Tools'
$ sudo dnf install wget texinfo PyYAML genisoimage flex bison
```

After installing dependencies for cross-compiler, we need to build cross-compiler toolchain for chosen architecture (in example amd64):

```
$ cd helenos/tools
$ ./toolchain.sh amd64
```

HelenOS is built using a meson system, therefore we need to install it:

```
$ pip3 install ninja
$ pip3 install meson
```

We need to create directory for our build (in source root for example) for chosen architecture (here amd64) and run configuration script:

```
$ cd helenos
$ mkdir -p build/amd64
$ cd build/amd64
$ ../../helenos/configure.sh amd64
```

After the configuration process, we need to use `ninja` to build HelenOS. After that we will have in directory helenos/build/amd64 image.iso, which we can use to start and run HelenOS, as described in the next section.

```
$ ninja
$ ninja image_path
```

To rebuild bootable image after code changes, we need to use these two commands.

A.2 Running

For standard start of the system it is enough to run `tools/ew.py` in the build directory for specific architecture after compiling source.

```
$ ./tools/ew.py
```

In the result the system is started with one default network driver e1000. Running `inet list-addr` will print:

```
$ inet list-addr
Addr/Width      Link-Name     Addr-Name Def-MTU
=============== ============= ========= =========
127.0.0.1/24    net/loopback  v4a       1500
...             ...           ...       ...
10.0.2.15/24    net/eth1      dhcp4a    1500
```

And the `pcapctl` application lists only one network driver capable of dumping packets.

```
$ pcapctl --list
Devices:
0. devices/\hw\sys\00.03.0\port0
```

If we want to run HelenOS with two network drivers we need to start system manually as `tools/ew.py` script does not support option for specifying several network drivers. Command for that is:

```
qemu-system-i386
-enable-kvm
-drive file=hdisk.img,index=0,media=disk,format=raw
-device e1000,netdev=n1
-netdev user,id=n1,hostfwd=udp::8080-:8080,hostfwd=udp::8081-:8081,
hostfwd=tcp::8080-:8080,hostfwd=tcp::8081-:8081,hostfwd=tcp::2223-:2223
-device ne2k_isa,irq=5,netdev=n2
```

```
-netdev user,id=n2,net=192.168.76.0/24,hostfwd=udp::8083-:8083,
hostfwd=udp::8082-:8082,hostfwd=tcp::8083-:8083,
hostfwd=tcp::8082-:8082,hostfwd=tcp::2224-:2224
-usb -device nec-usb-xhci,id=xhci
-device usb-tablet
-device intel-hda
-device hda-duplex
-serial stdio
-boot d
-cdrom image.iso
```

`qemu-system-i386` is in this example QEMU to start HelenOS for architecture `ia32`. For another architecture the name will slightly differ. Another necessary part of the command is redirecting second network driver to another network by specifying address `net=192.168.76.0/24`. Otherwise QEMU will assign the same address to both devices. That is a QEMU problem, not HelenOS (it was verified with starting Linux system).

After starting the system with that command `inet list-addr` and `pcapctl --list` will show that there are two network drivers and user can dump packets on any of them.

```
$ inet list-addr
Addr/Width       Link-Name     Addr-Name Def-MTU
================ ============= ========= =========
127.0.0.1/24     net/loopback  v4a       1500
...              ...           ...       ...
192.168.76.15/24 net/eth1      dhcp4a    1500
10.0.2.15/24     net/eth2      dhcp4a    1500

$ pcapctl --list
Devices:
0. devices/\hw\sys\00.01.0\ne2k\port0
1. devices/\hw\sys\00.03.0\port0
```

A.3 Running HelenOS with prebuilt image

In zip with the work there are two files image.iso (for ia32 architecture) and hdisk.img. The user can download them and in the folder where the files are run command (but first user must install QEMU):

```
qemu-system-i386
-drive file=hdisk.img,index=0,media=disk,format=raw
-device e1000,netdev=n1
-netdev user,id=n1,hostfwd=udp::8080-:8080,hostfwd=udp::8081-:
8081,hostfwd=tcp::8080-:8080,hostfwd=tcp::8081-:8081,hostfwd=tcp:
:2223-:2223
-usb
-device nec-usb-xhci,id=xhci
-device usb-tablet
```

```
-device intel-hda
-device hda-duplex
-serial stdio
-boot d
-cdrom image.iso
```

The command was run on Fedora 40 and MacOS 11.6.4 (M1).

Appendix B

User documentation for pcapctl

The chapter provides detailed user documentation and several examples of usage of the utility.

B.1 User documentation

The application is a console application that has required and optional options. User must specify the action: to start/to stop dumping or to list devices that can dump packets.

The options are as follows.

- pcapctl without any options will print the help page.
- List devices that can be used to dump packets.

```
$ pcapctl -l | --list
Devices:
0. devices/\hw\sys\00.03.0\port0
```

  The example demonstrates the list of devices capable of dumping packets after HelenOS was started with one network driver e1k (default network driver).

- Start the dumping for a particular device with particular writer operations.

```
$ pcapctl --start --device=0 --ops=0 --outfile=data.pcap
Start dumping on device - 0, ops 0.
```

  Or using short variants:

```
$ pcapctl -r -d 0 -p 0 -o data.pcap
Start dumping on device - 0, ops 0.
```

- Stop dumping on particular device.

```
$ pcapctl --stop --device=0
Stop dumping on device - 0.
```

  Or using short variants:

```
$ pcapctl -t -d 0
Stop dumping on device - 0.
```

- Utility enables using the default index of device and the default index of writer operations. Therefore, `--ops` and `--device` are not required options.

```
$ pcapctl --start --outfile=data.pcap
Start dumping on device - 0, ops 0.
$ pcapctl --stop
Stop dumping on device - 0.
```

- User-friendly short option to start dumping to new file with writer operations with index 0. The short variant is `-N`.

```
$ pcapctl --new=data.pcap
Start dumping on device - 0, ops 0.
```

- User-friendly short option to start dumping appending packets to the end of existing file (if file does not exist, it will be created). Short variant is `-A`.

```
$ pcapctl --append=data.pcap
Start dumping on device - 0, ops 2.
```

- User-friendly short option to start dumping truncated packets to file. Short variant is `-T`.

```
$ pcapctl --truncated=data.pcap
Start dumping on device - 0, ops 1.
```

- To overwrite existing file user can use option `--force|-f`.

```
$ pcapctl --new=data.pcap
Start dumping on device - 0, ops 0.
$ pcapctl --stop
Stop dumping on device - 0.
$ pcapctl --new=data.pcap
File data.pcap already exists. If you want to overwrite it,
then use flag --force.
$ pcapctl --new=data.pcap --force
Start dumping on device - 0, ops 0.
```

B.2 Examples of usage

Section provides several examples of how to use command-line utility. Another important detail is getting the PCAP file outside HelenOS. To get PCAP file outside of HelenOS user can start web server, copy file to web server root directory and get the file with `curl`, because the default HelenOS configuration sets up port forwarding with host machine.

Inside HelenOS the user runs:

```
$ websrv
websrv: HelenOS web server
websrv: Listening for connections at port 8080
$ cp data.pcap /data/web
```

Outside HelenOS user runs:

```
$ curl localhost:8080/data.pcap --output local_data.pcap
```

After that user can use wireshark to open file.

Typical examples of usage are listed below.

- Simple ping 8.8.8.8 to file.

  ```
  $ pcapctl --start --device=0 --ops=0 --outfile=data.pcap
  Start dumping on device - 0, ops 0.
  $ ping 8.8.8.8
  ...
  $ pcapctl --stop --device=0
  Stop dumping on device - 0.
  ```

- Starting HelenOS web server and dumping traffic. After starting web server, user needs to run "`curl localhost:8080`" outside of the HelenOS.

  ```
  $ pcapctl --start --device=0 --ops=0 --outfile=data.pcap
  Start dumping on device - 0, ops 0.
  $ websrv
  ...
  // running "curl localhost:8080" outside of HelenOS
  $ pcapctl --stop --device=0
  Stop dumping on device - 0.
  ```

- Running ping and dnsres after starting HelenOS with two network drivers. Appendix A explains how to start HelenOS with two network drivers. DNS and PING requests will be dumped to different files, because ...

  ```
  $ pcapctl --start --device=0 --ops=0 --outfile=data_0.pcap
  Start dumping on device - 0, ops 0.
  $ pcapctl --start --device=1 --ops=0 --outfile=data_1.pcap
  Start dumping on device - 1, ops 0.
  $ ping 8.8.8.8
  ...
  $
  $ pcapctl --stop --device=0
  Stop dumping on device - 0.
  ```

- Starting web servers on different ports after starting HelenOS with two network drivers and dumping traffic on both drivers.

```
$ pcapctl --start --device=0 --ops=0 --outfile=data_0.pcap
Start dumping on device - 0, ops 0.
$ pcapctl --start --device=1 --ops=0 --outfile=data_1.pcap
Start dumping on device - 1, ops 0.
$ websrv -p 8080
websrv: HelenOS web server
websrv: Listening for connection at port 8080
$ websrv -p 8083
websrv: HelenOS web server
websrv: Listening for connection at port 8083
// run "curl localhost:8080" and "curl localhost:8083"
// outside HelenOS
$ pcapctl --stop --device=0
Stop dumping on device - 0.
$ pcapctl --stop --device=1
Stop dumping on device - 1.
```

After getting files outside HelenOS and opening them with Wireshark, the user can see different addresses in two files from different network drivers.

Appendix C

User documentation for pcapcat

The application has several non-required options and required argument, which is file path. The options are as follows.

- `pcapcat` without any options and arguments will print the help page.
- User can specify maximal number of packets to be printed out with option `--count | -c <number>`

```
$ pcapcat -c 2 d.pcap
LinkType: 1
Magic number:  0xa1b2c3d4
0001) [IPv4] IP header: 20B, payload: 556B, protocol: 0x11,
Source IP: 10.0.2.2, Destination IP: 255.255.255.255
0002) [IPv4] IP header: 20B, payload: 264B, protocol: 0x11,
Source IP: 0.0.0.0, Destination IP: 255.255.255.255
```

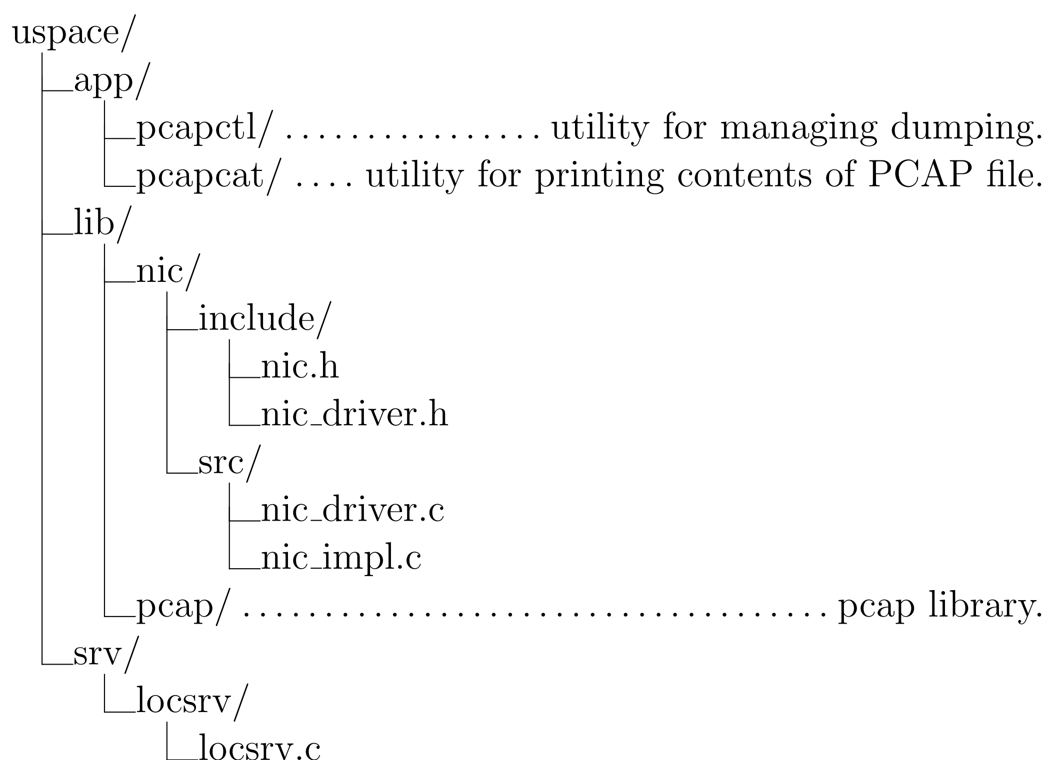
- The user can specify the level of verbosity with the option `--verbose | -v`. If the option is used, the application will print tcp ports of tcp segments.

```
$ pcapcat -v d.pcap
LinkType: 1
Magic number:  0xa1b2c3d4
0001) [IPv4] IP header: 20B, payload: 24B, protocol: 0x6,
Source IP: 10.0.2.2, Destination IP: 10.0.2.15
      [TCP] source port: 40828, destination port: 8080
0002) [IPv4] IP header: 20B, payload: 20B, protocol: 0x6,
Source IP: 10.0.2.15, Destination IP: 10.0.2.2
      [TCP] source port: 8080, destination port: 40828
```

Appendix D

Source code overview

Directory structure
Here is a list of modified directories and files.



In the nic library, little amount of code was added. However, added code enables every network driver present and implemented in the future to be able to dump packets.

Appendix E

Running CI tests locally

CI sub-project has its own project on GitHub (https://github.com/HelenOS/ci). Project with scenarios for our framework and new functionality can be cloned via `https://github.com/boba-buba/ci` and checkout branch topic/pcapctl-scenarios. To run tests locally, we need to clone repository (or use directory in zip with attachments) and in repository run test-in-vm.py with specified required options. Examples of commands:

Start HelenOS with the default configuration.

```
$ ./test-in-vm.py
--scenario scenarios/pcapctl/pcapctl_default.yml
--image path_to_image.iso
--arch [ia32|amd64|...]
```

Start HelenOS with two network drivers (e1000 and ne2k).

```
$ ./test-in-vm.py
--scenario scenarios/pcapctl/pcapctl_two_drvs.yml
--image path_to_image.iso
--arch [ia32|amd64|...]
--two_network_drvs
```

The commands were run on Fedora 40.

Appendix F

Programmer documentation

The chapter describes how to use an implemented framework to enable dumping packets on a chosen driver (which does not support dumping already).

1. Every driver in HelenOS has an associated DDF function. The DDF function is used, among other purposes, to add drivers to different categories. The concept of categories is explained in Section 4.1.4. Therefore, the programmer needs to add driver to category "pcap":

   ```
   errno_t err = ddf_fun_add_to_category(fun, "pcap");
   // fun is DDF function that is a part of driver structure.
   ```

   Usually, adding drivers to categories is done during initialization of the driver.

2. The programmer must add to the driver structure a new field of type `pcap_dumper_t` and initialize it during the initialization of the driver.

   ```
   errno_t err = pcapdump_init(dumper)); //dumper is pcap_dumper_t
                                           //field in driver
   ```

3. The programmer must add a function that dumps packets to a place in the driver code where the driver accepts and receives packets.

   ```
   pcapdump_packet
       (dumper, data, size); // dumper is a field of the driver structure
                             // data is a block of bytes
                             // size size of block in bytes
   ```

4. The last step is implementing `pcap_writer_ops_t` if framework does not have writer operations suitable for packets that go through the driver and choose them in command-line utility as `--ops`.