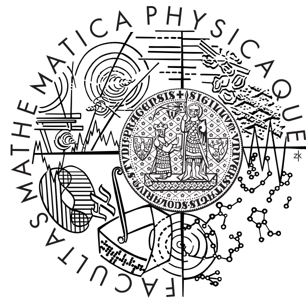
Charles University in Prague
Faculty of Mathematics and Physics

MASTER THESIS



Bc. Lukáš Mejdrech

Networking and TCP/IP stack for HelenOS system

Department of Software Engineering

Master thesis supervisor: Mgr. Martin Děcký
Study programme: Computer Science, Software systems

2009

Acknowledgements I wish to thank to my thesis supervisor, Mgr. Martin Děcký, for his advice and direction in this research. I owe the greatest gratitude to my family, for making this thesis possible and their constant patience and support.

I hereby declare that I have created this work completely on my own and used no other sources or tools than the ones listed, and that I have marked any citations accordingly. I agree with lending and publishing this work.

Prohlašuji, že jsem svou diplomovou práci napsal samostatně a výhradně s použitím citovaných pramenů. Souhlasím se zapůjčováním práce a jejím zveřejňováním.

In Prague Bc. Lukáš Mejdrech

Název práce: Networking a TCP/IP stack pro systém HelenOS
Autor: Bc. Lukáš Mejdrech
Katedra: Katedra softwarového inženýrství
Vedoucí práce: Mgr. Martin Děcký
E-mail vedoucího: decky@dsrg.mff.cuni.cz

Abstrakt: V této práci studujeme implementaci TCP/IP subsystému. Důraz je kladen na návrh a implementaci respektující koncept operačního systému s mikrojádrem. Praktickou částí pak byl vývoj TCP/IP subsystému pro systém HelenOS. Nejprve jsou popsány koncepty síťové architektury a TCP/IP subsystému obecně. Následují specifické aspekty kladené systémem s mikrojádrem. Dále je uveden návrh architektury a implementační rozhodnutí vlastní implementace.

Klíčová slova: síťové architektury, TCP/IP subsystém, HelenOS, ovladače síťového rozhraní

Title: Networking and TCP/IP stack for HelenOS system
Author: Bc. Lukáš Mejdrech
Department: Department of Software Engineering
Supervisor: Mgr. Martin Děcký
Supervisor's e-mail address: decky@dsrg.mff.cuni.cz

Abstract: Within this work we study networking and TCP/IP stack implementation. The main interest is directed to the TCP/IP stack design and implementation respecting the microkernel operating system concept. The practical part was a development of a TCP/IP stack for the microkernel operating system HelenOS. First, we describe the networking and the TCP/IP stack concept in general. The special aspects of the microkernel concept follow. For the practical part, the architecture design and implementation decisions are included.

Keywords: networking, TCP/IP stack, HelenOS, network interface drivers

Contents

1 Introduction **6**
1.1 Motivation 6
1.1.1 History 6
1.1.2 Implementation 7
1.2 Goals 7
1.3 Structure of the thesis 8
1.3.1 Style conventions 8

2 Context of the thesis **10**
2.1 Networking stack models 10
2.2 HelenOS specific design 12
2.2.1 Kernel code 12
2.2.2 Modularity 12
2.2.3 Inter–process communication 13
2.3 *Qemu* emulator 13
2.4 *N. E. T.* 14

3 Networking stack design **15**
3.1 New networking stack 15
3.1.1 Extent of the implementation 15
3.1.2 Architecture 16
3.1.3 Modular architecture 16
3.1.4 Packet management system 20
3.2 Networking stack modules 21
3.2.1 Central configuration module - *net* 22
3.2.2 Network interface drivers 23
3.2.3 Network interface layer 25
3.2.4 Inter–network layer 29
3.2.5 Transport layer 35
3.2.6 Application programming interface - *libsocket* 37
3.2.7 Applications 38

4 Discussion **40**
4.1 Implementation 40
4.1.1 HelenOS internals 40

4.1.2 Support structures 41
4.1.3 Modules 46
4.1.4 Startup module - *netstart* 54
4.1.5 Extending the networking stack 55
4.1.6 *Qemu* network 58
4.1.7 *N.E.T.* user protocols 58
4.2 Running and testing 59
4.2.1 Applications 59
4.2.2 Software prerequisites 61

5 Evaluation **62**
5.1 Nettest2 – data transfer performance 63
5.2 Nettest1 – the overall performance 67
5.3 Ping – ICMP echo performance 69
5.4 Conclusion 70

6 Other architectures **72**
6.1 BSD 72
6.2 Linux 73
6.3 Windows 74
6.4 Minix 74

7 Conclusion **76**

8 Terms and abbreviations **78**

Bibliography **81**

A Test results **85**

B UML diagrams **88**
B.1 Networking stack overview 88
B.2 Module parts interaction 90
B.3 Network interface initialization 92
B.4 Data transfers 94

Chapter 1

Introduction

1.1 Motivation

In this work we analyze, compare, design and implement a networking stack which is an operating system's subsystem connecting computers into networks. Although computer networks are in massive use these days there is no variability in reference implementations. Most of them are monolithic, having all the networking functionality bundled in one large module, although the stack is internally layered. The target implementation introduces modular architecture into the networking stack. This means building the stack up from smaller modules where each module encapsulates and provides one area of functionality, a protocol for example. They will be separate server modules each running as one task in the target operating system [1]. The stack is intended to be integrated to HelenOS, a microkernel operating system being developed at the Faculty of Mathematics and Physics, Charles University in Prague. The microkernel design and philosophy are briefly described as well as its requirements for additional modules' architecture. The result of this thesis makes effort to become a modular TCP/IP networking stack proof–of–concept implementation.

1.1.1 History

The history of computer networks began in late October 1969 when the ARPANET project was born [25]. Before that time computers were isolated individuals. Since that time they are able to communicate with each other.

The ISO/OSI model was developed to standardize computer communication and is the first and most comprehensive networking model. It defines seven layers of functionality and abstraction. Unfortunately the model does not contain concrete protocols and the applicants did not have anything to follow. Another problem was its megalomaniac attitude. The model tries to solve almost all possible questions and uses of interconnected computers. Therefore it was far to complex to implement and spread protocols respecting it. Furthermore, the applicants had to pay a significant amount of money just to obtain the detailed model specification.

[1] It can be viewed as a service oriented subsystem.

However, a different approach succeeded, the TCP/IP model. The model was based on simple and concrete protocols which were used to build up what was really needed. Its bottom–up approach led to its current world domination. This protocol family is called the TCP/IP protocol suite where the IP stands for the inter–network protocol and is the base protocol of the suite. It works on the principle of the best effort, the protocol tries to do its best to deliver data from one computer to another. The word "try" is the key as nothing is guaranteed. On the other hand TCP is probably the most used protocol on top of the IP protocol, it offers reliable connection between two computers. The TCP/IP suite contains many protocols covering most aspects of computer communication. Most of attention of this work will be drawn to this suite.

The protocols and design concepts are published as numbered RFCs which are public and rather technical documents covering usually only one topic, a protocol for instance. There is an RFC for each protocol in the suite, however, some protocols are extended in many RFCs. For example, there is one base RFC of the TCP protocol but hundreds of RFCs extending or altering its functionality in any way. RFCs do not become obsolete too often, they are usually updated or extended. The actual base RFCs of the TCP/IP suite come from early 1980s.

1.1.2 Implementation

As it was mentioned before, the TCP/IP protocol suite is extensively used. Computers usually do not have any other option how to connect themselves into current computer networks. One of the goals of this thesis was to develop a modular TCP/IP stack for a microkernel operating system HelenOS.

Porting an existing stack was also an option. It would have brought in many more features in exchange to the clean modular design as mentioned before. Some HelenOS-specific workarounds would have also looked odd in the system as a whole.

Therefore a new implementation is introduced. It is designed from scratch and based only on the relevant RFCs. The modular design and inter–process communication of the microkernel system are respected, of course. A modular and extensible architecture which seems to the author to be trusty to its purpose, as this architecture complies with the overall architecture of the target microkernel system. The TCP/IP protocol suite is very well described and documented in RFCs and almost no other resources are needed.

1.2 Goals

This work follows a few goals. The main goal was a networking stack implementation respecting the microkernel design of the target operating system. This emerged from the fact that the target HelenOS operating system did not have a networking stack. A new concept was to be thought up and designed. The networking stack moves HelenOS to the next level. It brings the possibility to communicate with other computers and systems.

The second goal is connected to the first one. This implementation might become, with a bit of luck, the reference implementation of a modular TCP/IP stack. Many believe that microkernel operating systems will have bright future and modular architecture is the right way to go. Probably the most famous propagator of the microkernel design is Andrew Tanenbaum, an author of many publications about operating systems [27] and the microkernel operating system Minix. There are many advantages and disadvantages, attitudes and reasons for and against both the monolithic and microkernel operating systems [2]. Most often mentioned disadvantages of the microkernel design are inefficiency and insufficient flexibility. Both are explained and at least partly invalidated in the work of Jochen Liedtke [9].

There is also a research capacity to compare the modular stack implementation overhead to the monolithic approach. For this purpose the compilation into many small modules or one large module is designed and supported. We will measure and analyze differences between these two approaches.

1.3 Structure of the thesis

Here is a description of the master thesis structure and content of single sections:

Chapter 2 An introduction to the context of the work, the networking stack model and HelenOS.

Chapter 3 A summary of the modular architecture requirements with the proposed networking stack design. It includes detailed networking model layers, analysis and proposed modules' description.

Chapter 4 A discussion about implementation decisions and problems.

Chapter 5 Architecture evaluation and comparison results.

Chapter 6 A few other networking stack implementations.

Chapter 7 A conclusion of the thesis.

1.3.1 Style conventions

The text follows a few style conventions:

- File and directory names are printed as `source_file.c`.
- All referenced header and source files are located in the HelenOS source sub directory `uspace/srv/net/` if not stated otherwise.
- Inter–process message names are printed as *MODULE_MESSAGE (argument)*.
- Code samples are printed in blocks

[2]Many were mentioned in the so–called Tanenbaum–Torvalds debate [1].

```
command;
function_call(argument);
```

- Constants and functions in the text are printed as ``constant'' and *function(argument)* respectively.

Chapter 2

Context of the thesis

2.1 Networking stack models

The networking stack can be divided into a few layers. Each of them represents an abstraction for a particular functionality. The ISO/OSI model [31] defines seven layers whereas the TCP/IP stack defines only four or five [26]. The TCP/IP stack layer count varies sometimes because the bottommost layer may be split into two. Although this work focuses on the TCP/IP stack the ISO/OSI model should be mentioned first as it is more general. Its concept uses seven layers. Each layer uses the layer underneath and provides some functionality to the upper one. Except some rare exceptions layers should not traverse more than one level [1]. The layer abstraction is that layers of the same level but on different hosts communicate with each other using lower layers transparently. A short description of the ISO/OSI layers follows and their scheme is in the Figure 2.1. The TCP/IP stack is described in detail the Section 3.2.

Physical layer

The bottommost is a physical layer. This layer transfers the smallest portions of data between network interfaces. Namely zeros and ones using electrical or optical impulses etc. Not only two network interfaces can be aware of the communication. Wireless networks are the best example that every network interface can hear the communication. Data unit being transferred by this layer is a bit. Some devices are capable to transfer more bits at once. In the sense of more bits at once, without any further knowledge about the data. This layer is about the transport medium, wires, air etc. and the transport perception.

Data link layer

The next layer is a data link layer. Whole data blocks are transferred between network interfaces. The source host directs the data block using the destination host's address. The addresses are physical, usually assigned by network interface manufacturers. The

[1] So called cross layer optimization [24].

	Data unit	Layer
Host layers	Data	7. Application
		6. Presentation
		5. Session
	Segment	4. Transport
Media layers	Packet	3. Network
	Frame	2. Data link
	Bit	1. Physical

Figure 2.1: ISO/OSI Layers, the colors are assigned to the layers here and entities in other figures will preserve the colors of their parent layers.

data blocks are called frames. The network interfaces are the main business of this layer.

Network layer

The third layer is the network layer. It provides logical addressing and routing of data blocks. The data blocks, called packets, are logically routed through networks by this layer. This layer makes a computer network allowing indirectly connected computers to communicate. The lower layers make connections only between computers on the same transport medium.

Transport layer

A transport layer is the next above. This layer provides a transparent data flow between two hosts. It can also provide reliable, flow and error controlled data transfers.

Session layer

The fifth is a session layer. It maintains logical connections between distant host applications. The logical connections allow the host to handle many networking applications at the same time.

Presentation layer

There is also a data presentation layer. Transferred data have to be interpreted in the same way on both communicating sides. There can be highly different devices on each end of the connection. Therefore some standardization or pre–transport negotiation is necessary. A network byte order is the best example. The big–endian byte order [2] is used in the networking.

Application layer

The last layer is an interface for applications willing to use the networking stack. It offers host identification, connection establishment and data flow to the end applications.

2.2 HelenOS specific design

In this section we describe the target system of the networking stack. HelenOS is under development of a research group at the Faculty of Mathematics and Physics, Charles University in Prague. It follows the microkernel design [8].

2.2.1 Kernel code

In operating systems there are two types of code execution, the kernel or privileged mode and the userspace or normal mode. Hosted applications run in the user space mode. The kernel code runs in a privileged mode. This mode allows access to all hardware, whole available memory including other tasks' memory and all parts of the operating system as well. Therefore any bug in the kernel mode may lead to system instability, data loss or even hang ups.

The smaller the code is the easier it is to double or triple check it. While studying the microkernel design, an interesting observation was made, that the kernel code can be really minimal [27]. The microkernel design attempts to minimize the kernel code and modularize the system as a whole. The kernel code needs to handle hardware interrupts, host applications and provide hardware access to drivers. HelenOS follows the thought that only really necessary parts should run in the privileged mode.

2.2.2 Modularity

As the microkernel design runs little code in the privileged mode it is easier to run more stable. In addition some operating system tasks do not have to run in the privileged mode at all. Many drivers or abstraction layers function well even when placed into the user space. The possible inconsistency due to code defects stays local in the task itself if it runs in the user space. The kernel basically gives no other option. Therefore much of the standard operating system functionality is broken into small pieces running in the user space. Such pieces can be called modules. They

[2] The most significant byte is stored first, with lowest address.

offer services to other modules so the overall architecture can be thought of as the service oriented architecture.

Furthermore the modular design may be used to reduce the overall complexity and ease development. The system is broken into functional components with defined interfaces [6]. The inner implementation is encapsulated and other components use it as a black box. The overall functionality is then built up from these components.

Another possible extension to the current view of an operating system is the possibility to check and restart defect modules. Either by polling the modules whether they are operational or by check–pointing their state.

2.2.3 Inter–process communication

Tasks communicate with each other through open connections using inter–process communication. The standard message passing mechanism is used. The sending task passes a message and can wait for a reply. The receiving task picks up the message and processes it replying the answer. Standard messages can have up to five integer arguments and up to five integer return values. The term "phone" is used in HelenOS for the task's identifier of an open connection.

HelenOS provides an asynchronous library which maintains these connections and creates a fibril for each of them. Fibrils are the smallest points of execution in the user space. They are like threads in a main thread. Each connection fibril processes messages only of its connection. This leads to parallelism where there can be many messages processed at a time. The client connections are isolated and do not interfere with each other.

A memory block can be also copied between two tasks. This is useful for large data blocks. The memory block is copied if the tasks agree. The kernel does the job in this case.

The last possibility is to share memory blocks. The kernel maps the memory block into the address space of the target task. So both tasks are able to use the shared memory. Great care has to be taken and the memory access needs to be synchronized or standardized.

2.3 *Qemu* emulator

As it is a bit hard to develop an operating system on a real hardware a simulator was used. A simulator gives the pleasure to develop test and debug the system in an efficient way. There is no need to dedicate a whole computer to run the system. Neither to transfer the built binaries to be able to execute them. *Qemu* is run on a host operating system and runs a guest operating system as if it was a standalone computer.

Furthermore, *Qemu* has a simple network interface. It emulates, among others, an ISA NE2000 network interface. ISA cards have the nice feature that the device IO port has to be statically set. This is the memory address used to communicate

with the device. Device registers are mapped there and the device is controlled by reading and writing them.

2.4 *N. E. T.*

In order to test and debug a networking stack a universal networking application is highly recommended. The stack itself contains functions and their counterparts which can be tested together. However, this would not reveal possible design mistakes, only programmatic. For example the checksum computation and check can function well together but it might be a different implementation than the protocol actually uses. Therefore an external application using another networking stack implementation is better as it tests these functions with their external counterparts. This should reveal both programmatic and design mistakes.

This application was developed as a testing tool of networking communication a few years before this work began. It is a modest application capable of communication using the lowest protocols, namely UDP and TCP. It is designed to provide useful networking communication information of the local and remote computers. The basic application features are:

- Sending and receiving packets of the TCP and UDP protocols on various ports as both the client and the server,
- Listening for connections,
- Ping and trace,
- Active ports enumeration,
- Getting host information including addresses,
- Getting protocol capabilities, configuration and information,
- Logging all printed information into a file, and
- GUI.

Despite the fact that this simple functionality is sufficient for the networking stack testing, the main feature is that the application allows users to define their own protocols. The whole process starting by a used protocol, connecting sequence, confirmation data, statuses and disconnecting sequence may be defined.

Chapter 3

Networking stack design

3.1 New networking stack

For the microkernel HelenOS a completely new networking stack was designed. The detailed description of requirements, architecture, support structures and modules follows.

3.1.1 Extent of the implementation

The networking stack was intended to implement current basic standards of the TCP/IP Stack. The standards are described in the form of RFC documents. The relevant RFCs are mentioned where appropriate. All the additional features of the TCP/IP Stack are far beyond the scope of this work. There are hundreds of extensions, alterations or concretions on top of the core design. Only the minimalistic functionality allowing the stack to function was to be implemented. The detailed functional rules are enumerated in later sections closer to their topics. The implementation goals were:

Initialize and use a real network interface network interface recognition, configuration, initialization and shutdown, sending and receiving packets, fault and state reception,

Support for more than one network interface advanced modular and dependency design, IP routing tables,

Implement the TCP/IP Stack IP, ARP, ICMP, UDP and TCP protocol modules,

Implement the BSD socket interface for applications a socket application library providing connecting and sending and receiving data functionality, and

Demonstrative applications ping, echo and similar applications.

The standardized TCP/IP Stack implementation allows the stack to coexist in most of the current networks whereas the standardized BSD socket interface eases porting of networking applications.

As HelenOS is written in the C language and it is an operating system with its own libc library, there are not many support structures. At least a basic object oriented approach is achieved by using structures as data classes and sets of functions manipulating them as methods. Implementation of the support of C++ in HelenOS would involve implementation of the C++ library. RTTI, inheritance, virtual methods, STL and integration with IPC would increase the scope of the thesis dramatically. Without this features there wouldn't be any additional benefits compared to C. So the networking stack is written in C.

3.1.2 Architecture

For the stack development two approaches are possible. One monolithic–like module and separate device drivers and on the other side a fully modular architecture can be used.

The monolithic–like module and separate device drivers have one big advantage – there is no internal inter–process communication at all. There are only normal function calls internally. The union of functional modules offers also the possibility to keep shared data on one address in the address space which decreases resource consumption. The performance is probably the most obvious reason to use this design. Nevertheless, there are disadvantages as well. Parts of this system are physically bound to each other. These parts correspond to functional modules in the meaning of reference model layers. Programmers are allowed to develop a bit fuzzy design with mixed global data structures and some possibly dirty workarounds and bypasses between functional modules. The stack is compact and extending it could involve internal changes. The stack cannot be divided easily either. Functional modules can depend on each other and function calls cannot be easily recognized and isolated. This is just a hypothesis if someone wants to exclude particular functionality. It can be, for example, an attempt to provide a special purpose networking stack based on a simplified one.

The fully modular architecture has every functional unit in a separate module. The modules communicate only using well defined interfaces. It needs a bit more detailed analysis of the modules' cooperation. Also the common data structures have to be distributed to all concerned modules. This approach is preferred in order to demonstrate a functional modular stack running in a microkernel operating system. There is one important advantage – each functional module is physically isolated. The functional modules have well defined interfaces and could be fully replaced without the need to modify any other modules. New modules are free to connect to this interfaces and extend the stack. There is also well defined module dependency and modules can be excluded.

3.1.3 Modular architecture

In this section we describe the module concept in general. The networking stack is split into many small modules which reduces the overall complexity and encapsulates functionality. Each module represents an actual implementation of a particular

protocol or driver. For each networking layer there are several modules. Every module communicates only in its or the next bottom or up layer. The modules provide their functionality only to their neighbours and the upper layer. It complies with the networking stack models where at most one level should be traversed. An overview can be seen in the Figure 3.1. Usually, networking stacks are internally modular. The difference is that the proposed one will be modular globally.

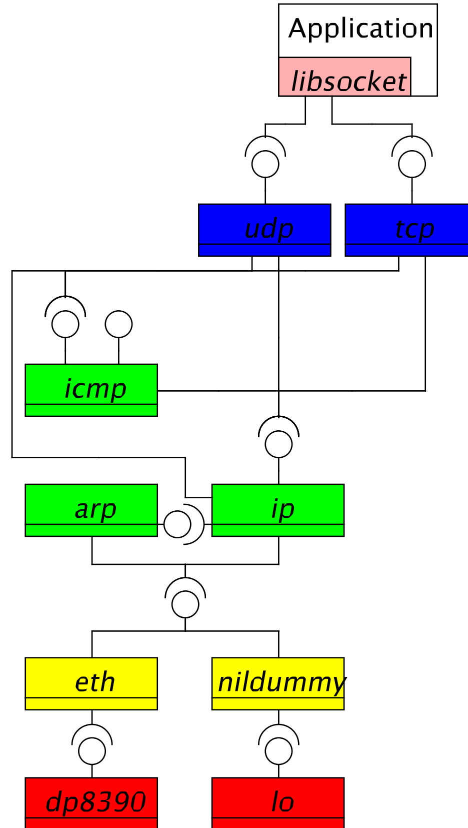


Figure 3.1: Modular networking stack

On one hand the networking stack architecture is a bit complicated, but, on the other hand it complies with the modular design. There can be several network interfaces controlled by their drivers. The TCP/IP stack does the routing and registers its receive points at the drivers. On the other end an application should not interact directly with the stack but rather through a standardized library. Applications use a networking library which provides an abstract socket interface masking the stack underneath. The stack itself contains many functional modules corresponding to the networking layers and protocols.

In the compile–time configuration of the proposed stack, either the modular or the monolithic architecture type can be chosen. This is supported with the consideration of the later demonstration and comparison of both the monolithic and the modular implementation.

In fact, the whole stack cannot be compiled into just one module as the network

interface drivers have to be separated. That is due to the big number of driver families and a networking stack bundled with all of them would be really huge. Therefore the right division line of the monolithic implementation has to be drawn. The same problem is a layer up in the network interface layer as well. There are many network interface access protocols which are specific to the used network interface driver family. For example Ethernet, Token Ring, ATM, Point–to-Point Protocol etc. They cannot control all types of network interfaces and vice versa. This means that the present network interface implicates the driver module and the data link module. On top of that, the TCP/IP suite works abstracting from the fact how the data link layer works. So the division is set between the data link layer and the network layer and is depicted in the Figure 3.2.

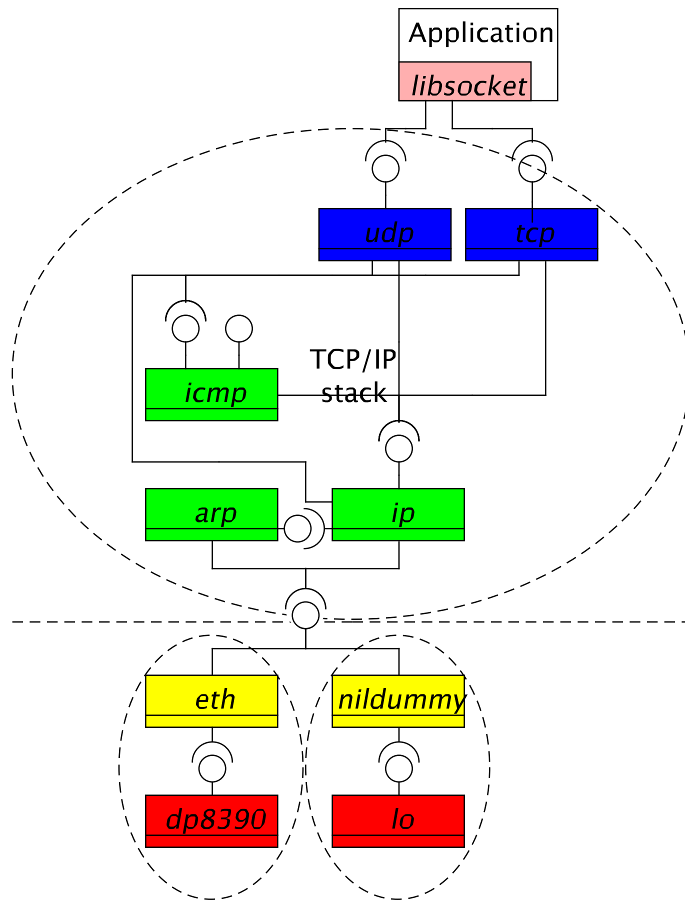


Figure 3.2: Monolithic networking stack

We should consider hardware and application limitations as well. There has to be a network interface driver for each network interface family. The driver can be also capable of controlling multiple devices in order to avoid multiple instances of the same driver and unnecessary overhead. On the other hand a bug in the controlling of one of the devices can break the controlling of the others. Devices keep most of their configuration and state in their own data structures and allow drivers work with

them through memory address spaces. Therefore the driver itself is almost stateless and it can handle more devices at a time without any disruption. There can be several network interfaces controlled by their drivers in the system at a time. Either using the same or several drivers [1]. The TCP/IP stack does the routing and registers its receive points at the drivers.

The applications willing to use the networking capabilities of the system use a library. The socket interface was developed to unify the abstraction of the networking stack. The applications using this interface can be ported to and function with other networking stack using the same interface. The library is only a wrapper providing the networking interface to applications.

Inter–process communication

In order to successfully orchestrate modules their cooperation should be carefully analyzed. There is an interface designed for each of the modules to publish its functionality. There are layer– and module–specific interfaces. The layer interface is a general layer functionality whereas the module specific serves its special purpose. This allows layer abstraction where more similar modules can work in the same layer and other layer modules could use either of them. The network and network interface layer modules are the best example as we discussed them in the context of the monolithic division line. Only known and almost hardwired connections would be possible otherwise. The driver just sends packets and state notification to its network interface layer buddy but the TCP protocol has to distinguish between the IP and the ICMP protocol. A module usually implements its layer interface.

Server modules should be able to serve many connections as they offer their services to others. They are also clients of other server modules. It would be very inefficient if a module had to wait until another message gets processed. Some complex messages can take a long time to complete. They can involve several other IPC queries. Therefore the networking stack should work in parallel.

On the other hand the parallelism lays stress on global data structures in the server modules. The global data have to be protected against multiple fibrils processing messages. Most of the messages can be easily divided into two groups. Some does not change the global data, but they need to read them. For example if a packet gets passed through a module, only the actual setup is read, the packet is processed and passed to another module. Other messages change the module setup. For example the initialization of a new protocol or device involves reading configuration or setting up routing tables. The most adequate synchronization primitives are therefore the read–write locks [2]. The rest of the messages can change the module setup on some conditions. They either swap the lock from reading to writing or lock it for writing. It depends on the likelihood of the writing process.

[1] An Ethernet router contains many instances of the same network interface to allow many hosts to connect whereas a host can contain an Ethernet and WiFi network interfaces.

[2] The read–write lock can be hold by multiple readers or by only one writer at a time.

3.1.4 Packet management system

A packet in the sense of the networking stack is a block of formatted data. Networking stack processes packets and have to pass them between its modules. The memory block can be passed between modules via inter–process communication messages. Unfortunately, this would be highly ineffective as the packet data would get copied due to the task isolation. In the monolithic–like stack this would happen only at the top and at the bottom of the stack, from client applications and to network interfaces. If the stack is to be modular the inter–process communication performance is the cutting edge. The fastest way is to transparently share the packet as an address space page. So only packet identifiers get transferred. This rapidly increases the stack performance. There is no internal copying of data. They are copied only at the top and at the bottom of the stack as in the monolithic case. However, the shared packet memory block has to be distributed between the concerned modules first. Strict rules need to be stated and obeyed as well:

1. Only modules in the networking stack and network interface drivers may have access to packets,
2. Only provided packet manipulation functions must be used,
3. A module processes a packet only if it is the exclusive owner of the packet:
    - The module becomes the exclusive owner if
        - requests a new packet, or
        - is asked to process the packet
    - The module is the exclusive owner until
        - passes the packet to another module, or
        - releases the packet.

The packet structure itself should never be accessed directly. Packet manipulation functions are provided in a form of a library described in the Section 4.1.2.

Architecture

The networking stack packet management system consists of a packet server and a library. The packet server needs to be implemented by one well–known module in the stack. Its job is to manage packets. All packets in the stack are in possession of this module. The packet library should be part of each module willing to process packets. It provides packet mappings into the task's address space and communicates with the packet server. Other modules can request a new packet and the packet server creates a new one or reuses an old one. Until the packet is released again it cannot be reused by the server. Any stack module can release the packet, not only the requesting one. If a module is asked to process a packet it uses the packet library to get the packet. If the packet has already been used, the library finds the packet mapping to the task's address space. If not, the library requests the packet from

the packet server. The packet server shares the packet as a memory block with the requesting task and the library returns the newly shared packet.

Packet processing

There are two directions of packet processing. A network interface driver uses clean or new packet to provide the received data to the networking stack. The transferred application data are incrementally unwrapped on their way through all the modules.

The topmost layer uses clean or new packets to send data through the networking stack. The transferred application data are incrementally wrapped on their way through all the modules. The processing time is crucial, so we would like to avoid data copying as much as possible. There has to be some infrastructure developed around the packet processing. As we mentioned before, each module wraps or unwraps the passed data. Each module knows how to do its job and does not have to be aware of the others. The idea is to keep the application data on one place during the whole process. The modules then incrementally just cut or prefix and/or suffix their specific wrappings.

Another option would be a list of succeeding memory blocks [3]. Each module just prefix or suffix its wrapping as another memory block. This increases the complexity as one packet is split into—possibly many—small pieces. For example a checksum computation of such a packet have to jump from one piece to another. Furthermore this is not a universal solution. All modules have to be able to process such block lists. However, some network interfaces are able to transfer only continuous block of data [4]. The blocks need to be serialized for them first. Unfortunately, this would be another data copying decreasing performance.

The packet size is loosely limited only by the MTU setting. However, the IP protocol, for instance, allows content only up to 65 535 octets long. The MTU may be specific for each device, network or protocol used. New outgoing packets have to be created with the MTU size taken into account. The IP protocol supports packet fragmentation for "small packet" networks. The upper protocols do not have to respect this setting as it is transparent for them. However, they could decrease load of their underlying IP module by doing so.

3.2 Networking stack modules

The HelenOS networking stack is described from the bottom layer up according to the TCP/IP model. The relevant ISO/OSI layers are referenced where appropriate. Each module description is followed by its public interface. Module interactions are described and some even depicted in UML diagrams. For a visual representation of the whole stack in an UML diagram see the appendix Section B.1. A central configuration module is described first.

---

[3] The technology of transferring and receiving succeeding blocks of data as one frame is called scatter–gather or vectored input–output.

[4] They can use only a simple DMA access to read a continues block without any other option, for example.

3.2.1 Central configuration module - *net*

The central module is the heart of the networking stack. This module implements the packet server, reads the networking configuration and starts required service modules. The configuration consists of the general networking stack and network interface specific parts. The stack configuration is textual. Numeric settings are parsed as integers, switches starting with $'y'$ as enabled. The general configuration specifies:

MTU The default maximum transmission unit. Set to `''1500''`.

ICMP_ERROR_REPORTING The Internet Control Message Protocol error generation and reporting switch. Set to `''yes''`.

ICMP_ECHO_REPLYING The Internet Control Message Protocol echo reply generation and reporting (*ping*) switch. Set to `''yes''`.

IPV The default Internet Protocol version. Set to `''4''`.

IP_ROUTING The general IP routing switch. Set to `''no''`.

UDP_CHECKSUM_COMPUTING The User Datagram Protocol optional outgoing traffic checksum computation switch. Set to `''yes''`.

UDP_AUTOBINDING The User Datagram Protocol optional automatic bind on send if not bound switch. Set to `''yes''`.

The network specific configuration contains:

NAME The network interface name.

NETIF The network interface driver module name.

NIL The network interface layer module name.

IL The inter–network layer module name.

The generic stack configuration can be overridden by the network interface specific. Any other configuration is also tracked so stack modules are free to add their specific settings. Module specific settings are mentioned at each of them. Modules can query the *net* module about a general or specific setting by its label. If the network interface specific setting is not found the general stack configuration is searched then.

This module starts needed modules of the stack and initializes network interfaces. The networking stack startup is described later in the Section 4.1.4.

net interface

The *net* module offers the following interface:

- *net_connect_module(net_service)* function and its backing *IPC_M_CONNECT_ME_TO* message to connect to the *net* module returning the connection phone.

- *net_free_settings(configuration, data)* to release the returned setting values. Should be used in conjunction with either general or network interface specific configuration request. If used in a bundle module, no memory is freed as the values buffer is empty.

- *net_get_device_conf_req(net_phone, device_id, configuration, count, data)*; *NET_NET_GET_DEVICE_CONF* to read the network interface specific configuration. The requested setting labels are transferred as measured string field and values in the opposite way. If used in a bundle module the values are not copied to the buffer and must not be freed.

- *net_get_conf_req(net_phone, configuration, count, data)*; *NET_NET_GET_CONF* to read the general configuration. Similar to the previous one.

3.2.2 Network interface drivers

A network interface, often referred as a network card, is the most important part of the stack. Roughly said, it is used to communicate with the outside world. Therefore all the outgoing and incoming traffic goes through the network interface. This is the bottommost layer in the ISO/OSI model, the physical layer.

There are a few basic capabilities of a network interface driver which need to be implemented. The most important ones are sending and receiving packets. The driver should be able to start and stop the device and accept a configuration from the networking stack. The devices can be present and configured but temporary disabled if desired. Controlling multiple devices is an optional feature.

Drivers can also provide some diagnostics and statistics for the system which should be updated appropriately.

The driver should be capable of:

- Discovering a device,
- Configuring a device,
- Starting a device,
- Stopping a device,
- Sending a packet,
- Providing a received packet,
- Returning device usage statistics, and
- Returning the current state of a device.

A network interface layer module can register itself as the packet supplier and consumer of the device. Incoming packets are delivered to that module asynchronously.

The network interface layer could also poll the driver frequently to obtain received packets. This option is not used as it increases overhead and is inefficient when there are no packets. On top of that if the driver is not queried often enough, the received packets can fill its buffers and may be discarded.

The driver can use device specific hardware settings such as input/output address space and IRQ. The driver then registers and handles IRQ events. Network interface drivers obtain the IRQ numbers and IO port addresses at the device initialization. Device configuration entries *IRQ* and *IO*, respectively, are forwarded to the driver.

Network interface driver interface

The network interface driver modules offer the following interface:

- *netif_bind_service(netif_service, device_id, calling_service, receiving_callback)* function and its backing *IPC_M_CONNECT_TO_ME* message to register itself as the received packets consumer.
- *netif_get_addr_req(netif_phone, device_id, address, data)*; *NET_NETIF_GET_ADDR* to read the network interface hardware address.
- *netif_probe_req(netif_phone, device_id, IRQ, io)*; *NET_NETIF_PROBE* to probe if there is a device at the specified memory address using the specified IRQ.
- *netif_send_msq(netif_phone, device_id, packet, sender_service)*; *NET_NETIF_SEND* to send the packet via the specified device.
- *netif_start_req(netif_phone, device_id)*; *NET_NETIF_SEND* to start the specified device.
- *netif_stats_req(netif_phone, device_id, stats)*; *NET_NETIF_STATS* to read the specified device's usage statistics.
- *netif_stop_req(netif_phone, device_id)*; *NET_NETIF_STOP* to stop the specified device.

Loopback network interface - *lo*

The loopback network interface is a special device which returns all outgoing packets back as incoming ones. The motivation for this device is to enable network communication between both server and client applications running on the same host. The applications are connected as being remote, however, the communication goes just through the local networking stack. There are no problems with carrier, protocols and formats and "the other side" is always reachable [5]. There is also no need of a real network interface device.

[5] Although the other application may not be reachable.

The *lo* driver is designed as a standalone module with capabilities of a regular device driver. The driver itself creates a virtual device and returns all outgoing packets as incoming incrementing the usage statistics accordingly. This is due to the requirement of usage statistics and transparent driver interface. In the extreme case only the user knows that this network interface is the loopback. Therefore the loopback does not need any special configuration.

NE2000 network interface driver - *dp8390*

The main goal of the networking stack is to connect the computer to a computer network. A real network interface enabling us to connect to the outside world is needed. There are hundreds of network interfaces and supporting them would be a long term run. For the successful networking stack demonstration one is enough. A NE2000 network interface family was chosen [6]. A lot of its clones [7] exists which is one of the main reasons. Another one is its simplicity. An ISA version of this network interface resides in the address space on one of the specified locations.

Qemu emulates the NE2000 network interface as well. So the decision was made to support this one. A more recent chip *DP8390D* by National Semiconductor actually. This chip is well documented in the data sheet [7] on the manufacturer's website.

3.2.3 Network interface layer

The network interface layer is the data link layer in the ISO/OSI model. This layer transfers whole blocks of data between two network interfaces, possibly on different hosts. The network interface layer module is specific for the type of network the computer is connected to.

There can be many network interface layer modules implementing various types of networks such as Ethernet, TokenRing etc. The network type is partly determined by the used network interface. Not all network interfaces support all types of networks. The NE2000 supports only the Ethernet.

The interface for the upper layers is transparent of the network type. So the bundle build is supported only with drivers, not the upper stack.

The network interface layer module can also distribute packets between the upper layer modules according to the inner frame protocol. There can be more than one upper layer module using this one for sending and receiving packets. The upper service module has to register itself and the network interface layer module can take into account the packet distribution. Packets are passed one by one. Concrete rules are module specific and will be described in the later example of *IEEE 802.3*.

Network interface layer interface

The network interface layer modules offer the following interface:

[6] This is a product line started in early 1990s originally by Novell. It became de facto a standard because it spread widely thanks to its low price.

[7] Clone is a compatible network interface, it behaves and can be controlled in a very similar way.

- For the upper stack:
  - *nil_bind_service(nil_service, device_id, calling_service, receiving_callback)* function and its backing *IPC_M_CONNECT_TO_ME* message to register itself as the received packets consumer, there can be more consumers per device based on the stated service.
  - *nil_device_req(nil_phone, device_id, mtu, netif_service)*; *NET_NIL_DEVICE* to register a device and its driver.
  - *nil_get_addr_req(nil_phone, device_id, address, data)*; *NET_NIL_ADDR* to read a network interface hardware address.
  - *nil_get_broadcast_addr_req(nil_phone, device_id, address, data)*; *NET_NIL_BROADCAST_ADDR* to read a network interface broadcast address.
  - *nil_packet_size_req(nil_phone, device_id, addr_len, prefix, content suffix)*; *NET_NIL_PACKET_SPACE* to read maximum packet dimensions, minimal address length, prefix and suffix and maximum content length in bytes.
  - *nil_send_msg(nil_phone, device_id, packet, sender_service)*; *NET_NIL_SEND* to send a packet (queue) via the specified device.
- For drivers:
  - *nil_device_state_msg(nil_phone, device_id, state)*; *NET_NIL_DEVICE_STATE* to process the device state change.
  - *nil_received_msg(nil_phone, device_id, packet, target_service)*; *NET_NIL_RECEIVED* to process the received packet.

Dummy network interface layer - *nildummy*

There is one special network interface layer module, a dummy one. It is used just to bridge communication between a network interface driver and the upper stack. One module is allowed to register itself as a consumer of received packets. Only devices registered with the *nildummy* module are available.

The purpose of this module is to keep the layer division. A driver does not have to process network interface layer messages nor sends them to an upper inter–network layer module. The messages are translated appropriately. Furthermore, the invariant that layers should not traverse more than one level is kept as well.

The most obvious network interface driver lying underneath is the loopback one where no added functionality is needed.

IEEE 802.3 - eth

One of the most widely used network types is the, informally called, Ethernet. The technology is standardized as *IEEE 802.3*. The history of the original Ethernet begun at XEROX PARC in 1976 [12]. The name comes from an ether, a fluid allowing

transfer of electromagnetic energy as it was thought at the end of the 19th century. The name Ethernet was released to the public domain later. It coexists as *Ethernet II* or *DIX Ethernet* [8] now. *IEEE 802.3* is a bit younger technology describing almost the same. *IEEE 802.3* covers data transfer between two network interfaces at the same medium, for example connected by a wire. There are a few variants according to the medium, transport direction and speed used.

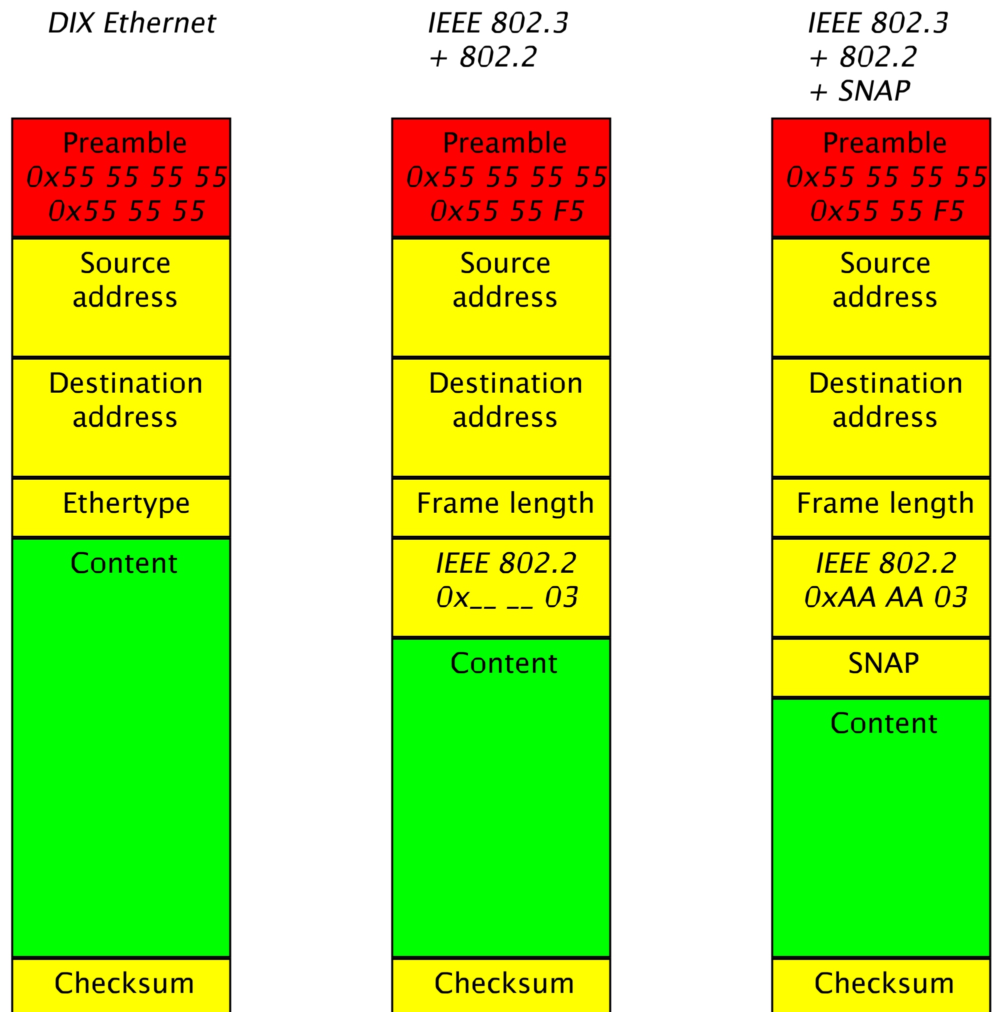


Figure 3.3: Ethernet frame formats

There can be also more network interfaces attached to the same medium and therefore addressing of data plays its role. Devices are assigned MAC addresses by their manufacturers. This addresses should be globally unique. They are hardwired but some devices allow users to change them. The network interface should receive only data labelled with its assigned MAC address [9]. There exists a standardized

[8]DIX stands for the founding companies Digital, Intel and Xerox.

[9]There are also network interface operation modes allowing to receive all frames or respecting an address pattern.

broadcast address to send data to all listening network interfaces.

The frame format differences are described in this paragraph and depicted in the Figure 3.3. Both *DIX Ethernet* and *IEEE 802.3* use almost the same header preceding the actual frame data. A frame type or ethertype header field in *DIX* is used as a frame length in *IEEE 802.3*. The basic frame type was used to distinguish the inner frame protocol, the higher protocol transferring its own data inside the Ethernet frame.

On the other hand the frame length offers better consistency check. Furthermore, the length is important as there is a minimum length limited. The frame content has to be at least 46 bytes long. It is padded with zeros if shorter. The frame length reveals such a padding whereas the frame type does not and the inner protocol or protocols have to find it out by themselves.

The *IEEE 802.3* does support frame type as well, but in an additional *IEEE 802.2* header. The usual frame types are all numbers higher than 1536 ($0x0600$). So IEEE decided to restrict frame type to be higher than 1536 and frames are allowed at most 1500 bytes of data. The coexistence of this two frame formats is explicitly possible. Frames can be easily recognized by the frame type/length header field.

The *IEEE 802.3* header can be further extended by *IEEE 802.2* and *SNAP* headers. The *IEEE 802.2* appends the destination and source service access point identifiers to the *IEEE 802.3* header. This service access points are equivalents to the frame types. The *IEEE 802.2* header format is in the Figure 3.4. The last byte—the

Figure 3.4: *IEEE 802.2* header, each field of one byte.

control one—can be used in many ways. For upper layers encapsulation only the Unordered Information frame type of the value $0x03$ is used.

When the service access points were designed a one byte field looked long enough. However, it was quickly almost exhausted. *SNAP*—represented in the Figure 3.5—is the most recent approach reintroducing two byte frame types on top of *IEEE 802.2*. Both service access points are set to $0xAA$ and the original frame type is attached.

Figure 3.5: *SNAP* header, organization or protocol code of three bytes, ethertype of two bytes.

The RFC 948 [28] defines transfer of higher protocol IP in Ethernet frames. The above mentioned frame types are supported. This definition can be also generalized for all higher protocols.

Ethernet also introduces a consistency check. At the end of each frame a checksum is carried. Ethernet checksum is based on CRC32. To be precise, a bitwise complement of CRC32 with a seed of all ones is transferred in the network byte order.This checksum is used to identify faulty transfers.

The *eth* module implements all of the above mentioned frame formats. A modern network interface should be able to receive frames of all formats and should generate only one frame format [16]. So *IEEE 802.3* with *SNAP* is the default despite the fact that the very first network interface (*DP8390* in *Qemu*) understands only the *DIX Ethernet* [10]. This emerged in a device configuration setting *ETH_MODE* of the `''DIX''`, `''8023_2_LSAP''` or `''8023_2_SNAP''` value. The *DP8390* network interface also handles frame preamble and checksum computation and check on its own. Therefore another device configuration setting is used to enable explicit frame preamble and checksum computation and checks. This switch is labelled *ETH_DUMMY* and is disabled by default.

3.2.4 Inter–network layer

Going a layer up, we traverse the conceptional division line. We are getting into the domain of the TCP/IP stack core protocols. The first layer supplies the ISO/OSI network layer.

This layer enables logical networking and routing. Packets being transferred between hosts can traverse multiple network interfaces or even logical networks. The inter–network layer allows hosts to identify available routes and other hosts' presence.

A host can have more network interfaces and this layer routes packets to appropriate ones. The host with more network interfaces can be also connecting more logical networks. They are so–called multihomed hosts and can function as gateways between the networks.

Inter–network layer interface

There is a standardized interface used by lower layers to communicate with the inter–network layer modules:

- *il_device_state_msg(il_phone, device_id, state, target_service)* function and its backing *NET_IL_DEVICE_STATE* message to process the device state change.
- *il_received_msg(il_phone, device_id, packet, target_service)*; *NET_IL_RECEIVED* to process the received packet.
- *il_mtu_changed_msg(il_phone, device_id, mtu, target_service)*; *NET_IL_MTU_CHANGED* to process the device MTU change.

On the other side no standardized interface exists for networking stack modules as they are aware of whom they are sending messages to. The TCP/IP Stack is designed

[10] In fact this is one of two proposed transfer methods for the IP protocol over the Ethernet in the RFC 948 [28].

to function with a concrete protocol in the inter–network layer, the IP protocol. So the upper layer modules do know exactly who is underneath.

Address Resolution Protocol - *arp*

As the TCP/IP stack communicates with hosts according to their logical IP addresses, lower layers do not understand them and the addresses have to be translated. The ARP protocol provides this service. The protocol is defined in the RFC 826 [21]. It is a universal protocol translating addresses to and from many protocols.

If an address translation is not already known a discovery request packet is sent to all connected hosts. ARP uses a variable length packet where appropriate hardware and logical address spaces are allocated for both the source and the destination host [11]. The destination hardware address is unknown and left empty. The hardware broadcast address is used as the destination address of the packet so other hosts' network interfaces receive the request.If the host recognizes its logical address a reply with all addresses filled is sent back to the request source.

ARP implementations cache translations for further use. If an error occurs—packets do not get delivered or the host unreachable notification is received—the cached translation should be cleared.

A timeout would be also an option. However, there is a bit of inefficiency. If a host is present, the timeout clears the mapping and the broadcast query needs to be send and replied. This puts unnecessary load to all connected hosts frequently. It can also lead to flooding in a big network. However, the timeout helps to discard unused entries and to propagate dynamic network changes.

On the other hand if a packet gets lost or an error is reported, the mapping is cleared and the packet can be resend. This involves the broadcast request as well, however, the caused load is unavoidable in order to recover from the error.

Although there is only the *arp* module present in the stack, some other can be added later. The particular ARP module can be specified via the device configuration *ARP* setting. This device specific setting is passed to the relevant inter–network layer module. That module should respect the *ARP* configuration and use the proposed ARP module.

ARP interface The address resolution protocol modules should offer the following interface:

- *arp_clean_cache_req(arp_phone)* function and its backing *NET_ARP_CLEAN_CACHE* message to clear whole cache.
- *arp_clear_address_req(arp_phone, device_id, protocol_service, address)*; *NET_ARP_CLEAR_ADDRESS* to clear the device protocol address translation.

[11] In case of IP and Ethernet the logical address has four bytes whereas the hardware has six bytes.

- *arp_clear_device_req(arp_phone, device_id)*; *NET_ARP_CLEAR_DEVICE* to clear the device cache.
- *arp_connect_module(arp_service)*; *IPC_M_CONNECT_ME_TO* to connect to the *arp* module returning the connection phone.
- *arp_device_req(arp_phone, device_id, protocol_service, netif_service, address)* *NET_ARP_DEVICE* message to register the *arp* module for the device at the driver and to save the requesting protocol logical address.
- *arp_task_get_id()* returning the task identifier if called in the bundle module.
- *arp_translate_req(arp_phone, device_id, protocol_service, address, translation, data)*; *NET_ARP_TRANSLATE* to translate the protocol address to the hardware address.

Internet Protocol - *ip*

The central protocol of the suite is the Internet Protocol. The IP protocol version 4 is defined in the RFC 791 [18] by DARPA from 1981. This protocol provides best–effort packet transfers. It tries to do its best to deliver packets to other hosts.

The fourth version is the first widely deployed version. There is a newer IP version 6. This version introduces longer addresses, extended addressing and routing, mobility, and IPsec. Implementation of this version could be a topic of another master thesis. The still used and simpler version 4 is implemented in this one instead.

IP addressing Hosts are identified by their IP addresses. Each address falls into an IP network. The address prefix of certain length is the network. The IPv4 uses four byte addresses and networks are assigned by IANA. There are groups of networks with fixed number of bits where the very first few bits of an address determine the network group. The textual addresses are written as four decimal numbers separated by dots.

The four byte addresses might have been enough for the world—it is 4 294 967 296 addresses, more than 4 billions—but not the network groups. An organization willing to have all its computers in its own network has to request a network. The network is assigned from the smallest group the computer count fits in. As bits in the address cannot be divided, only sizes in multiples of two are available. The problem deferred by using two special networks 192.168.0.* and 10.*.*.*. Anyone may use these networks while not being directly connected to other networks, especially to the Internet. Such networks are local and can be connected to other networks using gateways.

Along with network groups there are also network masks. It is a bit more universal approach. The address is divided into a network address and a host identifier. The mask contains ones at the network address places and zeros at the host identifier, respectively. A network can be divided into smaller parts if the mask sets more bits.

In order to communicate with others, a host has to maintain its routing table. The table contains entries where to send packets targeted to other hosts. Hosts of the

same network can be accessed (almost) directly but hosts of other networks not. The institution of multihomed gateways is introduced for the latter case. The host can be configured to send packets to other network using a gateway. The gateway forwards that packets further to the proper network or another gateway. The gateway routing can be set as a network and a netmask, where only the mask bits are compared to the target address. It can be less than the target network mask. Similar entries can be grouped together by shortening the mask. This is called the Classless Inter–Domain Routing.

Hardware address resolution Any ARP module can be configured for the hardware address resolution needs. The *ip* queries the ARP module for IP address translations then.

There was an option to ask the ARP to send the packet itself which would save one message in the inter–process dialogue. However, it would shift some IP functionality to ARP modules which would be against the modular architecture decided before. Therefore the address is requested, read and the packet is then sent. Please see Figures 3.6 and 3.7.

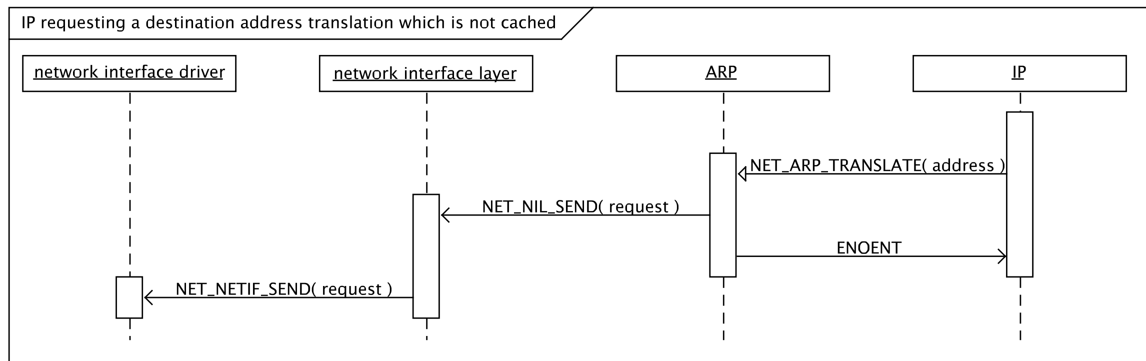


Figure 3.6: IP requesting a destination address translation which is not cached.

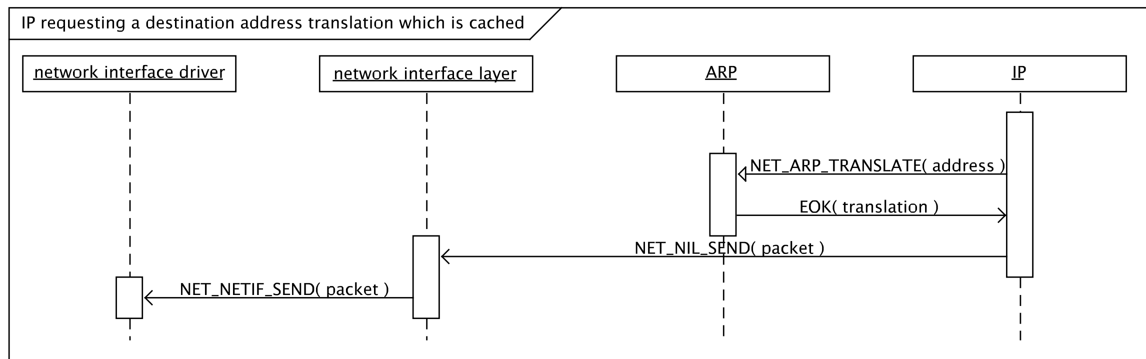


Figure 3.7: IP requesting a destination address translation which is cached.

IP protocol The IP packet header consistency is verified by a checksum. Although this checksum is weaker than the CRC in the Ethernet Section 3.2.3 presented earlier, it is used in other core TCP/IP protocols as well. The RFC 791 [18] authors stated

> ...experimental evidence indicates it [the checksum] is adequate, but it is provisional and may be replaced by a CRC procedure, depending on further experience.

Nevertheless, the proposed checksum computation is still in use, 28 year later. This is partly because it is sufficient and partly because it has spread widely and any change would be very costly as many implementations already use that. Its RFC definition [12] seems hard to understand but the principle is in fact very simple. The packet header is divided into 2 byte blocks which are summed. The result is split into high and low two bytes and summed again until the high part is zero.

As some protocols are not obligated to compute the checksum and set it to zero a computed zero checksum has to be distinguishable. The computed zero is flipped into all ones.

On one hand there is problem of this checksum as it cannot recognize 2 byte blocks being swapped, but, on the other hand it allows partial checksum computation and, furthermore, in any order. This feature and more are presented in the RFC 1071 [4] with implementation examples to speed up the checksum computation.

***ip* module** The *ip* module implements the IP protocol. The module supports more network interfaces and implements CIDR routing tables.

Based on the device configuration *IP_ROUTING* switch it can serve as a router. The incoming traffic will be forwarded further if not targeted to the local host, the router. For security reasons only routers should enable this setting. An Internet attached host can serve as a gate to its inner local network otherwise, even if not intended to. Packets for the inner network sent from the outside network would be routed properly in.

The module also allows connection to the local host via a loopback network interface which should have the IP address 127.0.0.1. The default gateway for undetermined routing direction can be set as well. The IP protocol supports packet fragmentation if a packet does not meet the target network MTU.

In case of errors during packet processing, error notifications are generated using the ICMP protocol [13]. Either if the network is unreachable, the host is unreachable, the checksum check fails or the time to live runs out. No additional notifications should be generated while processing error notifications.

Client modules register themselves at the *ip* module by their IP protocol identifiers. Received packets are delivered according to these protocol identifiers. The networking stack can be easily extended with other protocols on top of the IP protocol.

---

[12]The checksum computation and optimization techniques are described in the RFC 1071 [4] and its updates 1141 [10] and 1624 [22].

[13]The ICMP protocol is described in the later section as well as general error processing.

IP interface The *ip* module uses more device configuration settings along with the *ARP* and the *IP_ROUTING*. First and most important is the *IP_CONFIG* denoting `''static''` address configuration. No other value is supported yet as the networking stack contains only core protocols and further would be necessary, namely DHCP or at least BOOTP. For the static configuration the network interface address *IP_ADDR*, *IP_BROADCAST* and *IP_GATEWAY* addresses and *IP_NETMASK* are read. They should be set to a textual IP addresses, `''10.0.2.15''` for example.

***ip* interface** The *ip* module offers the following interface:

- *ip_add_route_req(ip_phone, device_id, address, netmask, gateway)* function and its backing *NET_IP_ADD_ROUTE* message to add a route entry for the device.
- *ip_bind_service(ip_service, protocol, calling_service, receiving_client_connection, receiving_callback)*; *IPC_M_CONNECT_TO_ME* to register itself as an upper protocol, either the callback or the client connection function is used according to the build architecture.
- *ip_connect_module(ip_service)* to connect to the *ip* module returning the connection phone.
- *ip_device_req(ip_phone, device_id, netif_service)*; *NET_IL_DEVICE* to register a device and its driver.
- *ip_get_route_req(ip_phone, protocol, destination, address_length, device_id, pseudo_header, header_length)*; *NET_IP_GET_ROUTE* to get a destination direction and pseudo header for sending.
- *ip_packet_size_req(ip_phone, device_id, addr_len, prefix, content suffix)*; *NET_IL_PACKET_SPACE* to read maximum packet dimensions, minimal address length, prefix and suffix and maximum content length in bytes.
- *ip_received_error_msg(ip_phone, device_id, packet, target_service, error_service)*; *NET_IP_RECEIVED_ERROR* to announce a received error notification.
- *ip_send_msg(ip_phone, device_id, packet, sender_service, error_service)*; *NET_IL_SEND* to send a packet (queue). A particular device can be specified. If the error service is set, no further errors are generated.
- *ip_set_gateway_req(ip_phone, device_id, gateway)*; *NET_IP_SET_GATEWAY* to set the default gateway.

Internet Control Message Protocol - *icmp*

The ICMP protocol defined in the RFC 792 [19] is a support protocol for the TCP/IP suite protocols. It carries error notifications and diagnostic messages. If an error occurs an ICMP packet is created. It contains the beginning of the original packet starting with the original IP header.

ICMP packets are sent via IP and there is a protocol number assigned for the ICMP protocol as well. Therefore ICMP behaves in a similar way like upper protocols in the transport layer. The *icmp* module registers at the *ip* module and received ICMP packets are then delivered to the *icmp*. However, the ICMP protocol actually falls into the inter–network layer next to the IP protocol. It does not add extra logic as other protocols in the above layer.

The ICMP error reporting can be disabled by setting the global configuration *ICMP_ERROR_REPORTING* switch to anything else than `''yes''`.

The ICMP protocol offers more than just error notification. There is one highly demanded functionality, the echo. One host sends an echo request to another host and waits for a reply. If the other hosts is reached—and is not configured to block ICMP echo requests—the same packet is sent back as the echo reply. The packets contain sequence numbers to match the replies with the previous requests. The ICMP echo replying can be disabled by the global configuration *ICMP_ECHO_REPLYING* switch. A local application is allowed to connect to the *icmp* module and call the echo process.

The ICMP protocol client application can use the interface:

- *icmp_connect_module(icmp_service)* function and its backing *IPC_M_CONNECT_ME_TO* message to connect to the *icmp* module returning the connection phone.

- *icmp_echo_msg(icmp_phone, message_size, timeout, ttl, tos, dont_fragment, address, address_length)*; *NET_ICMP_ECHO* to ping a host.

***icmp* interface** The *icmp* module offers the following interface:

- *icmp_connect_module(icmp_service)*; *IPC_M_CONNECT_ME_TO* to connect to the *icmp* module returning the connection phone.

- *icmp_destination_unreachable_msg(icmp_phone, code, mtu, packet)*; *NET_ICMP_DEST_UNREACH* to report the destination unreachable error.

- *icmp_parameter_problem_msg(icmp_phone, code, pointer, packet)*; *NET_ICMP_PARAMETERPROB* to report the parameter problem error.

- *icmp_source_quench_msg(icmp_phone, packet)*; *NET_ICMP_SOURCE_QUENCH* to report the source quench error.

- *icmp_time_exceeded_msg(icmp_phone, code, packet)*; *NET_ICMP_TIME_EXCEEDED* to report the time exceeded error.

3.2.5 Transport layer

Although the IP protocol offers host to host communication this is not enough. The next layer, the transport layer, enables more host connections to run parallel.

Every connection or connection–less transfer uses ports. The host address, protocol and port determines the communication source or destination. This triplet is called a half–socket. The other side is also identified by a half–socket, both together creating the socket. The socket is used to identify the communication.

Client application create and hold such half–sockets. The client part of the art is described in the next Section 3.2.6.

Transport layer protocols have to register themselves at the *ip* module with their IP protocol number. After that they can use the *ip* interface and consume their received packets. They can also use the ICMP protocol if desired.

The transport layer is the topmost layer of the networking stack. Client applications are directly connected to modules of this layer using the—later described—socket library.

As this is the last layer of the stack no packets are allowed to go any further. Client application data have to be copied to and from their address space from and into packets.

Transport layer interface

There is a standardized interface used by lower layers to communicate with the transport layer modules:

- *tl_received_msg(il_phone, device_id, packet, target_service, error_service)* function and its backing *NET_TL_RECEIVED* message to process the received packet. Either the registered function or the IPC message is used.

Internet Control Message Protocol - *icmp*

As the ICMP protocol falls into the inter–network layer in fact it is described there in the Section 3.2.4 earlier. It supports the general transport layer interface.

User Datagram Protocol - *udp*

The User Datagram Protocol is the simplest transport layer protocol. It is described in the RFC 768 [17] and does not add any other functionality to the underlying IP protocol but sockets. Client applications are offered the state–less and best–effort communication. Data can get lost, arrive out–of–order or more than once.

The module can be configured to compute checksums for outgoing packets. The same checksum as in IP is used except that it spans the IP pseudo header, the UDP header and the data as well. The IP header is included in the checksum to identify misrouted datagrams and the UDP header and data are included to verify their consistency. The checksum is optional but highly recommended. The UDP checksum global configuration switch is called *UDP_CHECKSUM_COMPUTING*.

The UDP protocol is a state–less protocol so it can be viewed as half–sockets communicating with each other. An application can send data to anyone and receive data from anyone. The application has to provide the destination identification, the

address and the port. With a received packet the source address and port are always provided with the packet data.

Although it can be desired to just send data and not receiving any, the default is not to. The module assigns a listening port automatically on send if not already assigned. This behaviour can be changed using the *UDP_AUTOBINDING* global configuration switch.

Transmission Control Protocol - *tcp*

The second transport layer protocol is much more powerful. The TCP protocol establishes reliable communication between two hosts and ensures data transfers. Both sides cooperate to deliver all data exactly once and in the right order.

The hosts establish the connection first. One side listens and waits for an incoming connection as the other one initiates the connection. Similar dialogue happens at the end of the connection.

Transferred data are acknowledged when received in the next outgoing or a separate packet. Packets are retransmitted if not acknowledged within a specific time. The TCP protocol uses the same consistency checksum as the UDP protocol. The *tcp* module contains much more logic maintaining socket states and buffering data. The comprehensive description of the TCP protocol can be studied in the work of Zaghal and Khan [30].

There exist many extensions on top of the original RFC 793 [20]. To name some of them: congestion detection and avoidance, MD5 checksum, slow start, fast retransmit, selective acknowledgement, adaptable retransmission timers etc. Many are enhancements, optimizations or security improvements.

3.2.6 Application programming interface - *libsocket*

There is a client library for client applications willing to use the networking capabilities of the system. A standardized—in the right meaning—API does not exist. However, the BSD Unix operating system came up with a networking client design in the release 4.2 BSD in 1983. Its application programming interface became a de facto standard soon and is called the Berkeley socket interface [11].

The client library implements core functions of the Berkeley socket interface. This fulfils networking needs and eases porting of networking applications. The library maintains local sockets and connections to needed transport layer modules and registers its own client connection function to process messages sent by these modules. All this is transparent to the client application.

***libsocket* interface** The *libsocket* library offers the following interface:

- *accept(socket_id, address, address_length)* and its internal *NET_SOCKET_ACCEPT* message to get an accepted socket and the remote host identification.

- *bind(socket_id, address, address_length)*; *NET_SOCKET_BIND* to bind the socket to a local port for listening.
- *closesocket(socket_id)*; *NET_SOCKET_CLOSE* to close the socket. This function behaves like the original *close()* but differs in the name as the name is used in HelenOS for file operations.
- *connect(socket_id, address, address_length)*; *NET_SOCKET_CONNECT* to connect to the remote host.
- *getsockopt(socket_id, level, name, value)*; *NET_SOCKET_GETSOCKOPT* to get the socket option value.
- *listen(socket_id, backlog)*; *NET_SOCKET_LISTEN* to set the maximum number of waiting accepted sockets.
- *recv(socket_id, data, length, flags)*; *NET_SOCKET_RECV* to receive the data via the connected socket.
- *recvfrom(socket_id, data, length, flags, address, address_length)*; *NET_SOCKET_RECVFROM* to receive the data and the remote host identification via the listening socket.
- *send(socket_id, data, length, flags)*; *NET_SOCKET_SEND* to send the data via the connected socket.
- *sendto(socket_id, data, length, flags, address, address_length)*; *NET_SOCKET_SENDTO* to send the data via the socket.
- *setsockopt(socket_id, level, name, value)*; *NET_SOCKET_SETSOCKOPT* to set the socket option value.
- *socket(domain, type, protocol)*; *NET_SOCKET* to create a new socket.

All the functions except the *closesocket()* are blocking—do not return until completed.

3.2.7 Applications

To demonstrate the operational networking stack a few applications were implemented. The both presented are common part of operating systems with networking stacks.

Ping - *ping*

The first application's purpose is to test if the networking stack is operational. It tests whether a host is reachable by sending an ICMP echo message. If testing the localhost address `‘‘127.0.0.1’’` the loopback network interface is checked. If configured

correctly the *icmp* module sends, replies and receives the echo message reporting success. If testing another host's address the real network interface is checked instead. The *icmp* reports success if a reply to the sent request is received. The application prints the overall time taken in both cases.

Echo - *echo*

The second application listens at a specified port and echoes back what it receives. The application prints the received data and the source address and port.

Chapter 4

Discussion

4.1 Implementation

The implementation follows the networking stack design from the previous sections to fulfil the TCP/IP stack needs. In this section implementation details and optimizations are described. This part is rather brief and technical.

4.1.1 HelenOS internals

Some HelenOS specific implementation details are mentioned in this section. These are only relevant fragments of the concept of the system from the networking stack point of view.

Services A name server module where all tasks can register their services themselves is used in HelenOS. The concept can be looked at as a service oriented operating system where any task can be a service. Even more services. They are used as system unique identifiers of the module capabilities.

Other modules can query the name server if they are looking for a particular service. The name server redirects them to the registered module if there is such.

This gives another layer of abstraction and allows indirect task connections, where the querying task does not know which task it needs to connect to.

The networking stack modules are registered as such services as well. The services are defined in the `uspace/lib/libc/include/ipc/services.h` header file in the HelenOS source directory.

Parallelism If a module wants to process multiple connections as in our case almost everywhere in the stack, an asynchronous manager provided in HelenOS serves the best. Module starts in one fibril in one thread and calls the *async_manager()* function after initialization. This function starts the asynchronous manager and runs further in the fibril. There is a new fibril created for each incoming connection. Due to this fact many fibrils can be present in the service (server–like) modules.

Device numbers Devices in HelenOS should be assigned a device identifier. This identifier enables a system–unique identification of the device. The identifier is generated in the kernel via a system call *device_assign_devno()*. The system unambiguity of the identifiers can be achieved only in the kernel.

Network interfaces are devices as well and therefore the networking stack requests an identifier for each present. Modules in the stack refer to them using these identifiers.

The central configuration module is able to assign a virtual device number to the present network interfaces. These identifiers would be networking unique but not system unique which could lead to confusion. Furthermore, the usage of the provided system call complies with the system designers' attitude.

4.1.2 Support structures

A few structures to support the development are going to be introduced. Some of them are template–like data structures for use with any data types. There are simple containers to fulfil basic needs of the networking stack in its first version as it is more a prototype rather than a tuned–up performance–optimized stack.

packet structure

The packet is a memory block with a header at the beginning and a data container afterwards. This memory block is aligned to the address page size. Packets are also extended to carry their addresses to ease inter–process communication. Modules can use packet address containers to pass addresses. The packet is designed as a shareable memory block with the header containing:

- The packet identifier, the unique packet identifier assigned by the packet server,
- The packet queue placement, packet queue sort order and metric as well as previous and next packet identifiers,
- The packet length in bytes, the overall packet structure and data container size,
- The packet source and destination addresses, universal space for passing addresses with the packet between modules, they are stored right after the packet header and the data container starts after them,
- The addresses length, the actual length of the stored addresses,
- The prefix reserved bytes, the number of reserved bytes before the application data,
- The data reserved bytes, the number of reserved bytes after the application data,

- The data start offset, the actual data start to be used at the next processing step,
- The data end offset, the actual data end to be used at the next processing step, and
- The data container.

The packet header structure is defined in the `structures/packet/packet_header.h` header file. Only the packet server and the packet client library should include and use it. There is only masked packet pointer definition for client modules *packet_t*.

Packet server

The packet server is the central authority of the packet management system. Only one module in the networking stack should implement it. Its implementation is in the `structures/packet/packet_server.c` source file. The hosting module should be the first and the last running of the whole stack and has to forward all packet messages to the *packet_server_message()* function. The packet messages are defined in the `structures/packet/packet_messages.h` header file. New packets are created with the view of memory sharing. Therefore the requested size is rounded up to the address page. The task receiving a memory block always allocates whole address pages anyway. So the packet uses as much space as possible which does not only decrease overhead but also eases packet reuse.

The packet server keeps released packets in queues according to their total size. There are seven sorted queues to distribute free packets. The first one contains packets up to one address page size. The second up to two address pages size and so on exponentially. Packet is inserted to the lowest queue it fits in. New packet requests are primary served from this queues. The best fit and last recently used algorithm is used which tries not to waste resources and to avoid frequent TLB page faults. The queues are sorted and searched from the first of a sufficient packet size. If no free packet is found a new one is created. If an old packet is reused it is probable that at least some of the processing modules have already used it. If they have, they have got it shared in their address space. Therefore the packet server might not be queried to share the packet at all. The older the packet is the higher is the probability.

Packet library

Every packet management system client module needs to use the packet library. This library takes care of packet mapping, requesting from and releasing to the server transparently. The client module has to provide only the packet server phone. The library also offers functions to manipulate packets such as:

- Set and get packet addresses, variable memory blocks up to the reserved address size,
- Copy data into the container,

- Prefix and suffix data returning a pointer to the new area allocated,
- Queue and detach a packet to and from a packet queue, and
- Traverse a packet queue.

For the packet mapping a global data structure is used. It is an integer to pointer map with a safety lock. As packets are numbered in the ascending order starting from one and are not released until the module terminates, only a simple map is used. The map is created at the module start and keeps filling up with time. It never releases a packet mapping until destroyed. Mappings are divided into pages of one hundred identifiers each. So the actual mapping position may be obtained instantly (in the $O(1)$ time) using the following macros:

```
/** Packet map page size.
 */
PACKET_MAP_SIZE                  100

/** Returns the packet map page index.
 *  @param packet_id The packet identifier.
 */
PACKET_MAP_PAGE(packet_id)    \
(((packet_id) - 1) / PACKET_MAP_SIZE)

/** Returns the packet index in the corresponding packet map page.
 *  @param packet_id The packet identifier.
 */
PACKET_MAP_INDEX(packet_id)   \
(((packet_id) - 1) % PACKET_MAP_SIZE)
```

The map pages are allocated only if needed as the total number of packets in the system is variable and is not distributed to all modules. If the packet map is destroyed it releases all the mapped packets. The packet server itself is also a packet client because it needs the packet mapping as well to remember all the existing packets. The only difference is that the packet server inserts the mapping itself and does not query anyone.

The last optimization is queuing packets. Modules are able to send the whole queue as if it were only one packet. If possible a packet queue is the preferred way as it decreases processing overhead.

Measured strings

Another inter–process communication structure is a measured character string. This is a universal structure containing a character string pointer and its length. The structure is the measured string header and the actual string is the measured string data. The structure is used universally for any type of memory block not just the character string. Its purpose is to transfer large data blocks between modules.

There are support functions to send and receive arrays of such strings transparently. The original motivation was to read configuration from a central module. There are four variants as HelenOS distinguishes between the initiator (i.e. the client) and the supplier (i.e. the server) for each data flow direction. The communication sequence is very simple. The measured string field size has to be transferred in the parent message and then the fields may be sent/received supplying the size. The sending function computes and sends the lengths of all strings in the field because both sides have to agree on a length in order to transfer a data block. The receiving reads the lengths and allocates sufficient memory blocks for both headers and data. The strings are then transferred one by one as they may not be in one continues block in the sender's memory. The receiving function puts them one after another into the data block separated by the null ($'\backslash 0'$) character. The receiving module then gets the headers field and the data block. After processing both may be just freed. The other direction is the same with different HelenOS functions.

This is a workaround to increase performance as well. Standard messages can take only up to five arguments and both sides have to agree on data lengths. The strings are of variable lengths so the lengths have to be transferred first. Another option is to use the parent message arguments but they may not be enough. So the additional transfer of lengths is needed. This also exhausts one message argument if the modules are not already aware of the number of strings. On the other hand this approach is less efficient for just one measured string as the length could be carried in a message argument instead of the field size. In this case the lengths transfer is extra.

Dynamic FIFO queue

This structure is a first–in–first–out queue with dynamic expansion. It uses a round integer buffer which may expand if full. Insert function contains a maximum size parameter to limit the queue size. The queue expands only up to this parameter if specified. If an expansion occurs the buffer is reallocated preserving the inserted values so the expansion may involve copying all values to the new buffer. The expansion can be viewed in the Figure 4.1.

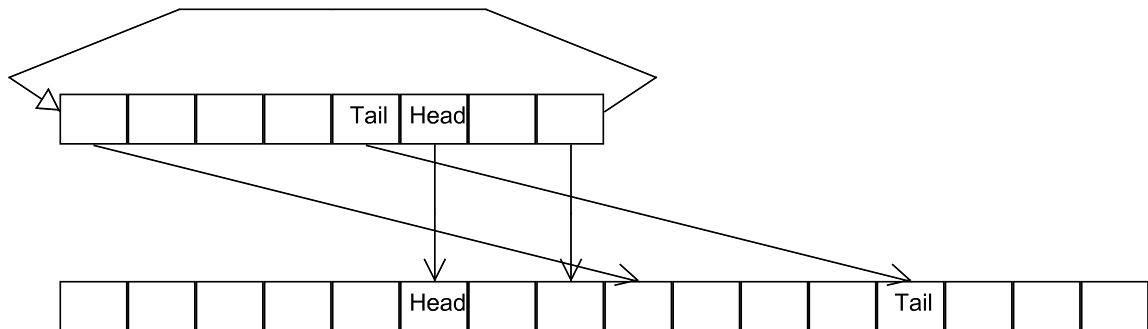


Figure 4.1: Dynamic fifo queue expansion

Character string map

The character string map provides a mapping between character strings and non–negative integers. It provides functions for inserting, updating and excluding values. Values are stored in a tree structure. Each node represents a character and a value for a string starting at the root node. Each node contains also a field of child nodes according to the next character. The field is not a significant slow down for only sparse maps used in the stack. The character string may be extended by its length in which case the map functions traverse null characters ($'\backslash 0'$).

Integer map

This is the first macro data structure. It maps integers to pointers of any type. The map has to be declared and implemented by a set of two macros

```
/** Integer to generic type map declaration.
 *  @param name Name of the map. Input parameter.
 *  @param type Inner object type. Input parameter
 */
INT_MAP_DECLARE(name, type)

/** Integer to generic type map implementation.
 *  Should follow declaration with the same parameters.
 *  @param name Name of the map. Input parameter.
 *  @param type Inner object type. Input parameter
 */
INT_MAP_IMPLEMENT(name, type)
```

with the same names and types. These macros define functions prefixed by the `name` accepting and returning pointers of `type`. If a value is inserted to the map the map becomes the exclusive owner of the value. The values are freed when excluded from the map or when the map is destroyed. The map returns an assigned index to a newly inserted value for further extending or looping. It uses a flat field of key–value pairs and therefore the map does not sort values and search time is linear (in the $O(n)$ time). It should be stated that the map is used only for small sets of values such as device identifiers.

Generic field

The generic field is similar to the integer map but contains only values without keys.

Generic character string map

This is a compound data structure containing a character string to integer map and a generic field as represented in the Figure 4.2. Indexes to the generic field are stored in the character string map. It maps character strings to pointers of any type.

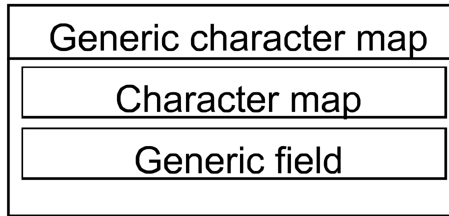


Figure 4.2: Generic character map

Module map

The module map serves as a container to store information about networking modules. It is a generic character map to a *module* structure containing a task identifier, a service identification, a module message phone, an optional usage counter, module name and filename and a module connection function. The purpose of the map is to unify a service modules' usage. A module fills the map with possible service modules it can use at the startup. The module calls a function to retrieve a running service module later. The map checks if an entry exists, if the module is running[1] and if the module is connected [2]. If not it starts (spawns) the module and/or connects to it transparently.

4.1.3 Modules

The networking stack modules' implementation details are described in this section. The programming documentation is in the `doc/` directory generated by Doxygen using JavaDoc style comments. The general module concept is described first.

Module design

The networking stack was decided to support both modular and monolithic compilation. The networking stack architecture can be set by the compile–time configuration option called *Networking architecture*.

Each service module has to implement handling of client connections, processing of IPC messages and its public interface and specific core functions. While the bundle module implements handling of client connections, processing of all IPC messages and public interfaces and specific core functions of all its modules. Therefore the stack modules are compounds of many parts in order to support both modular and monolithic build.

The module parts are located in separate source files to avoid any architecture specific source code variants. Furthermore, the source code is distributed into many smaller functional units which reduces the code complexity. For a visual representation of the module parts assembly see the Figure 4.3. The module parts interaction in UML diagrams the appendix Section B.2.

[1]The task identifier would be zero.
[2]The phone would be zero.

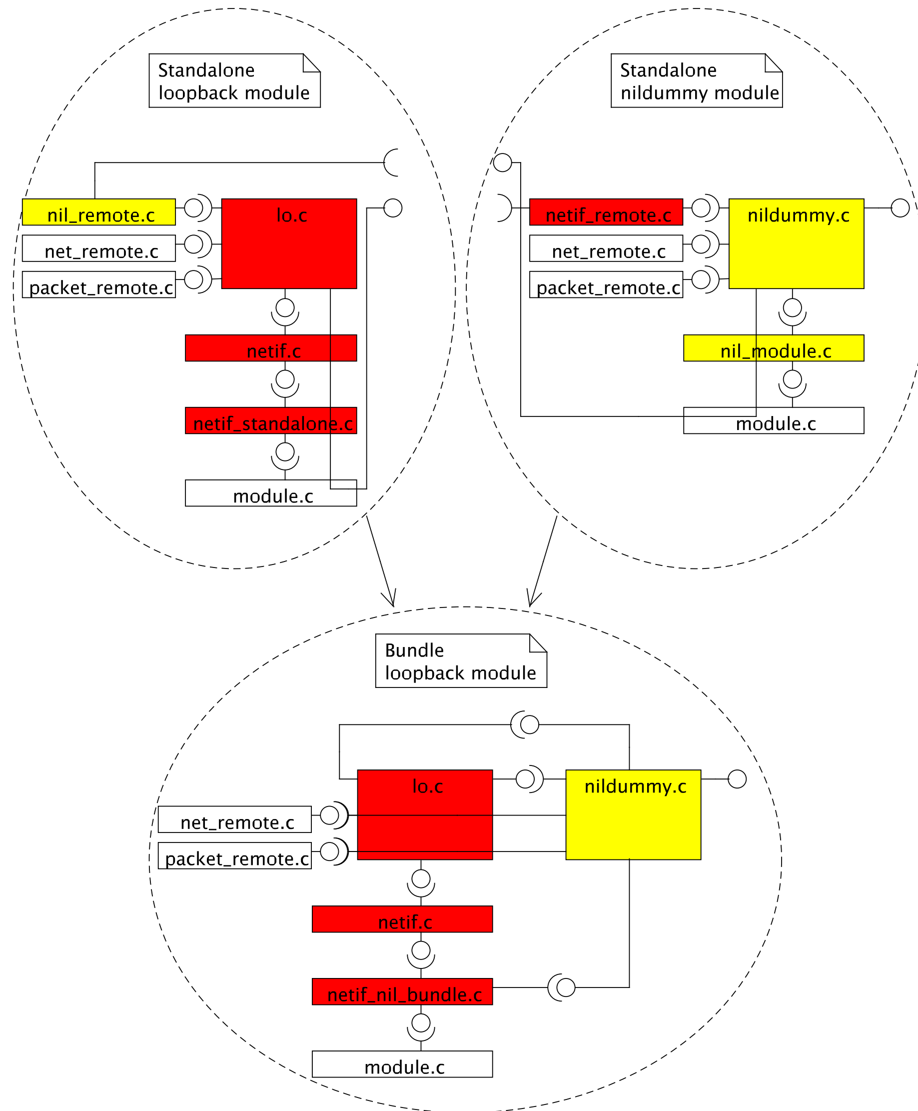


Figure 4.3: Standalone and bundle module assembly

Module parts There is a module skeleton in the `module.c` source file. This skeleton starts a module by the module's startup function and forwards inter–process messages. The startup function should initialize the module, register the skeleton's client connection function and start the asynchronous manager.

The module stubs are located in `(module_name)_module.c` like source files containing the startup and message procesing functions. In a standalone module they just call their module specific variants. While in a bundle they initialize all bundled modules by their initialization functions and distribute IPC messages to appropriate module specific functions. All services the bundle offers should be also registered.

The public interface is implemented in the module itself and in a remote client source file. The remote client files are named like `(server_module_name)_remote.c`. This implementation forwards the interface function calls to the module as standard

IPC messages. In order to use the interface all client modules have to be built in the bundle with the module or standalone with the remote file.

Interfaces and public constants are located in the `include/` directory. Some more often used interface functions are statically defined in the `messages.h` header file.

As the networking stack can be built as one big module, the message destination has to be uniquely specified. All bundled modules share the same connections so the target service is added as one of the message arguments. According to this argument the real destination module in the bundle can be identified.

The module core implementation source file `(module_name).c` contains the module specific startup and message processing functions, the public interface implementation and the specific core functions. The message processing function maps the public interface to the internal interface or core functions. This means that there are in fact two entry points for each inteface function, a message from the message processing function and a direct interface function call.

Compilation As there are no architecture specific source code variants in separate functional units, the compilation was designed to build each source file at most once. Object files `.o` representing `.c` source files are created and the target architecture is achieved while linking built object files together and assembling modules.

If a module is built as a standalone the skeleton and this module stub files have to be included. If it is a part of a larger module, only the module core implementation source file is used. The main module's bundle startup file and the skeleton file are included.

When the other build architecture is chosen no files have to be rebuild which increases the compilation performance. Only the object files are assembled in a different way into proper modules.

Built modules are then transferred into the `/srv/` directory in the HelenOS boot image.

Packet/packet queue processing

In this section we describe the general concept of packet passing. Although the packet and packet queues are interchangeable they can have different meanings.

Network interface The network interface driver can send packet queues. The packets are transmitted one by one. On the other hand the received packets can be buffered and sent as a packet queue to the network interface layer module.

Network interface layer Packet queues can be passed to the network interface modules from both directions. Each packet means exactly one frame. They are processed and can be forwarded as packet queues as well.

Inter–network layer The *arp* module can receive packet queues and processes the packets one by one.

The *ip* module is designed to process received packets or packet queues separately. Due to the possible fragmentation and the attempt to minimize data copying, a fragmented packet is stored as a packet queue containing the packet fragments. Therefore if a packet queue is transferred to the upper layer, it is also only "one" packet. On the other side packet queues from the upper layer have to be fragments of one large packet as well. This is an optional optimization described in the next section.

Transport layer Client application data have to be copied to and from their address space from and into packets. Data copying is an expensive operation, talking about both time and resources. With this taken into account an optimizing approach is designed.

If a packet is received its size in the number of packets (fragments) is sent to the appropriate application. When the application initiates the data read, block of data is continuously read from the packet fragments.

On the other side the transport layer module can specify maximum data fragment size. When the client application initiates the data write, data are written in blocks of this size. The transport layer module writes these blocks right into packets.

The fragment size can be the maximum protocol data length. Such packets can get fragmented later which would involve data copying. If the module is aware of the network interface's MTU and subtracts necessary prefix and suffix lengths, great savings can be obtained. The application data get copied right into packet fragments which will be passed through the stack without any more copying.

net

The networking stack configuration is static for each of the stack architectures. The configuration is in the source code as HelenOS did not have file operations support. File operations and disk image mapping are young features in HelenOS and are not used by the stack so far. The general design decision about configuration files in HelenOS is not made yet.

The module source is located in the `uspace/srv/net/net` sub directory of the HelenOS source directory. There are two stack architecture specific source files, `net_standalone.c` for the modular and `net_bundle.c` for the monolithic build.

Network interface driver

Architecture The driver module is designed as a skeleton and a stub implementing device specific routines. The reason to do it like this is that this universal driver design allows bundle build with any of the network interface layer modules. Furthermore, the driver is the main module in the bundle.

The skeleton is located in the `netif/netif.c` source file and uses the general module skeleton source file `module.c`. The driver stub implements an interface defined in the `netif/netif_module.h` header file bacause the network interface skeleton forwards messages to these functions. The skeleton provides some common func-

tionality like message handling (not processing), keeping device information, safety locking, module initialization and packet mapping and releasing.

Device states Device states are defined as *device_state_t*. A device is enabled in the *NETIF_ACTIVE* state. Device drivers might be, but should not be, started at the boot. They should be started by the *net* module on demand which saves some resources until needed.

Usage statistics Network interface usage statistics are provided in a form of the *net_device_stats* structure.

IRQ An IRQ recognition command sequence can be registered in the kernel. Only a small set of safe commands [3] is used. This sequence is run by the kernel if an interrupt occurs. It should check its device state and accept the interrupt if caused by the device. The kernel then calls the user space callback function. An example is in the *DP8390D* section later.

***lo* bundle** The networking module attaches the loopback driver to a dummy network interface layer module. The *nildummy* module just passes all messages to and from the driver. Therefore it is much more efficient to combine networking interface layer and driver capabilities into one module and save a bit of inter–process communication and system resources. There is support for a separate module as well as a bundle module. In a productive system the bundle loopback module is recommended as it lowers both the overhead and resources.

dp8390 Although the chosen chip *DP8390D* is well documented writing a driver from scratch is out of the scope of this work. Furthermore a real hardware is needed at least at the final stage of the driver development as simulators' functionality can slightly differ. Both reasons lead to porting an existing driver from other systems.

There are not many drivers under a suitable licence [4]. The Linux network interface driver design is a bit complicated as it spans over many source files. On the other hand Minix does not support so many network interfaces and drivers are easily recognizable. On top of that Minix uses the same licensing as HelenOS does so the Minix driver was taken and ported into HelenOS. The Minix driver uses four probing functions to support the four most common clones of the basic NE2000, however, only the original NE2000 was ported.

There are only a few settings the NE2000 ISA card allows and the *Qemu* card is located at the address $0x300$ and uses IRQ 9.

In order to map an address space of a device into the task's address space the *pio_enable()* system call is used. With this function a device driver is able to directly

[3] IRQ commands are defined as *irq_cmd_type* in the `kernel/generic/include/ddi/IRQ.h` file in the HelenOS source directory.

[4] Only a few licences allow (almost) free usage and distribution. HelenOS is under the BSD licence.

control its device registers. The probe function is called next to verify the right device's presence.

When the device is started an IRQ handler has to be registered via the system call *ipc_register_irq()*. This function registers the IRQ recognition command sequence and a callback function. The command sequence is run in the kernel whenever an interrupt occurs. For the *DP8390* the following sequence is used:

1. *CMD_PIO_READ_8* reading an interrupt status register,
2. *CMD_PREDICATE* skipping the next command if the read value is zero, and
3. *CMD_ACCEPT* accepting the interrupt, if not skipped.

When the interrupt is accepted the kernel sends an IPC notification which calls the callback function. The interrupt recognition command sequences are run in the kernel mode with interrupts disabled whereas the callback function runs in the user space mode with interrupts enabled again. When an interrupt occurs there can be more devices using the same interrupt number and therefore they have to check if it is really their interrupt being processed.

This shows that only the recognition command sequence is run in the privileged mode. The kernel checks the proposed command sequence and referred addresses while registering it. This driver is a true user space driver as none of its code [5] runs in the kernel mode.

The driver module in the `netif/dp8390/dp8390_module.c` source file is written to provide a solid base for other network interface drivers. There is the network interface stub interface and the interrupt handler implemented. The ported network interface core functions in the `netif/dp8390.c` source file are accessed via a specific interface in the `netif/dp8390_drv.h` header file.

Network interface layer

Interfaces The `nil/nil_remote.c` source file contains functions called only by drivers whereas the upper stack has to send IPC messages in all cases. As the upper stack interface is based on common message functions defined in the `messages.h` header file, no additional `..._remote.c` file is needed.

Bundle within a network interface driver The network interface layer module can be built bundled within the network interface driver module. The driver is the main module and hence should initialize the network interface layer core at the startup. The driver registers itself, not the network interface layer service, and the upper stack should connect directly to it. This is due to the lack of the universal network interface layer functionality as the bundle cannot function with another driver except the bundled one. So the networking stack has to be configured to use or not the network interface layer.

[5] The interrupt recognition command sequence is in fact a kernel script.

The standalone network interface layer modules should be present and available at all times—despite the chosen networking architecture—as some additional drivers may not be bundled with their network interface layer modules.

***eth* checksum** Several variants of the CRC32 function are supported as it was not clear at the beginning which one is the actually used one. It is the CRC32 in the host endian version with bit inversions at the beginning and at the end.

```
htonl(~ compute_crc32(~ 0u, start, length_in_bits));
```

IP client modules

Because the IP protocol header can contain options and an ICMP error notification has to contain the original IP header, the header is transferred with the packet to the upper layer. The client module itself decides whether it wants to read or set some IP options or not. Client modules should also prepare packets to be sent by calling the *ip_client_prepare_packet()* function. The IP header should precede only the first packet fragment of the queue.

The *ip* module offers also a so–called IP pseudo header. This basic header contains only source and destination addresses, the upper protocol and the data length. It is often used by client modules for security reasons or checksum computation [6].

The IP protocol client modules can use IP support functions defined in the `include/ip_client.h` header file:

- *ip_client_get_pseudo_header(protocol, source_address, source_length, destination_address, destination_length, data_length, header, header_length)* to build the IP pseudo header.
- *ip_client_header_length(packet)* to get the received packet's IP header length.
- *ip_client_prepare_packet(packet, protocol, ttl, tos, dont_fragment, ipopt_length)* to prepare packet to be sent via the IP protocol.
- *ip_client_set_pseudo_header_data_length(header, header_length, data_length)* to update the stored IP pseudo header with the data length.

ICMP error processing

If the *ip* module wants to create an error notification it checks if the *icmp* module has registered. The *ip* module is operational even without the *icmp* module but no error notifications are generated in that case. Other modules just connect to the *icmp* module.

The packet and the error description are sent to the *icmp* module, as depicted in the Figure 4.4. The problematic packet is truncated, put into the ICMP packet and sent as a normal packet via IP. In addition, the error service is set to the *SERVICE_ICMP*. This is to avoid generating of error notifications for error notifications.

[6] See UDP in the Section 3.2.5 for details.

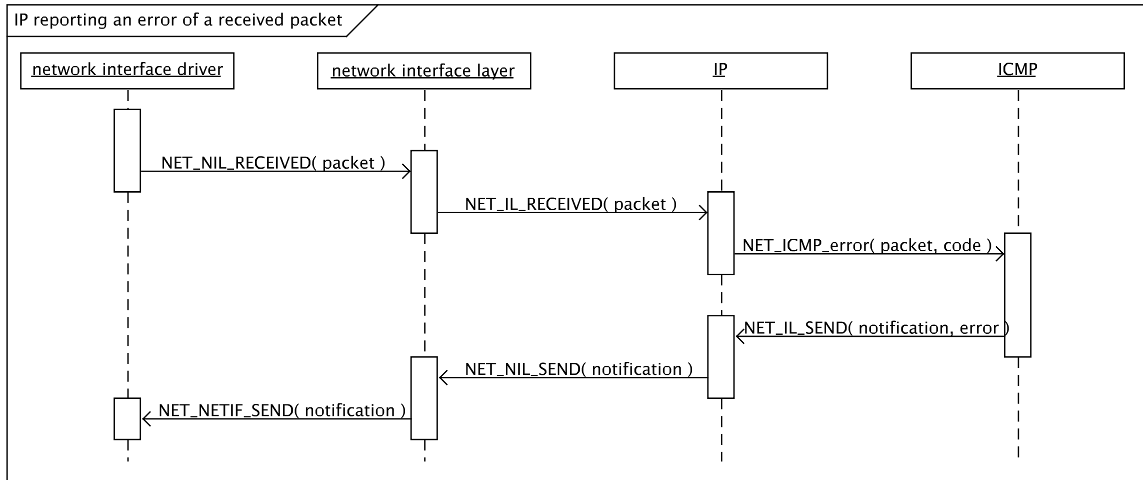


Figure 4.4: IP reporting an error of a received packet.

On the other side if an error notification is to be received the packet is delivered to the *icmp* module first. The notification IP header is dropped and the packet with the ICMP header and beginning of the original packet—starting with the original IP header—is returned back to the *ip* module. The error service is set and the packet is to be delivered further as a fault packet. The original protocol receives the packet and is notified that it is reported as faulty stating the *SERVICE_ICMP* error service. If the upper protocol understands the error service it can extract the error description. It can also drop header up to its own—the ICMP and the original IP one's in our case—and try to identify where the error occurred. The process is in the Figure 4.5. The error notification process is universal so other error handling protocols can be

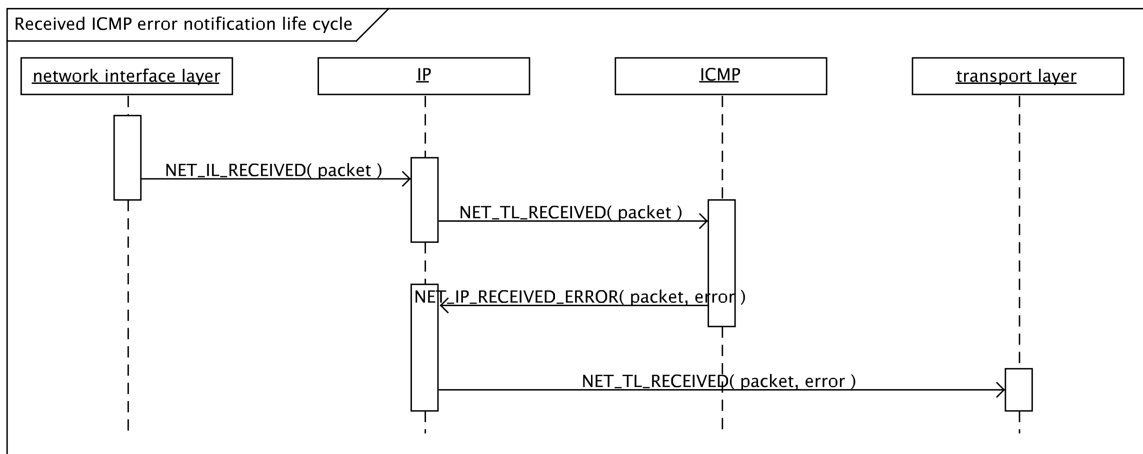


Figure 4.5: Received ICMP error notification life cycle.

added later. The TCP/IP stack uses only the ICMP though.

The ICMP protocol client modules can use ICMP support functions defined in the `include/icmp_client.h` header file:

- *icmp_client_header_length(packet)* to get the ICMP header length of the faulty

packet.

- *icmp_client_process_packet(packet, type, code, pointer, mtu)* to extract error description form the faulty packet.

The ICMP protocol client application interface is defined in the header files `include/icmp_api.h` and `include/icmp_common.h`.

Transport layer

The transport layer contains two modules, *udp* and *tcp* implementing their protocols above the *ip* module and under the socket library. The *udp* module is fully functional and tested for use whereas the *tcp* module is a prototype yet and is going to be completed and extended after the HelenOS team revise the networking stack architecture and specify their detail requirements.

Sockets

The common part of the transport layer modules is to maintain existing half–sockets and communication with client applications. The client applications implement the client socket library which handles the networking communication in fact. The socket functionality is moved into the socket core functions whereas other common features are left in the transport layer. The socket core functions help maintain existing sockets, bound half–sockets and transfer data between the applications and the transport layer modules.

The sockets are designed to buffer received packets and accepted sockets while waiting for the application to retrieve them.

Socket client library The library should be able to be statically or dynamically compiled with applications. The socket types, domains and address and protocol families are defined in the `include/socket_codes.h` header file whereas the possible return codes are defined in the `include/socket_errno.h` header file. The library *libsocket* is build as part of the networking stack. The built socket library is located in the `socket/libsocket.a` file.

4.1.4 Startup module - *netstart*

The *net* module is started by its buddy *netstart* module in the `net/start` directory. This small module can run some self tests of support structures and starts the networking stack. *net* initializes itself and waits for a *NET_NET_STARTUP* message sent by the *netstart*. This is so that the module can be started with a delay. This can be useful if moved to the initial tasks of HelenOS and disabling the network interfaces' automatic start.

During the networking startup network interfaces' configuration is read. The stack has two pre–configured, a loopback interface and a NE2000, both are described in the Section 3.2.2.

The configured modules are initialized for both. The driver task is started and the probe message from the networking module activates device probing. On success the device should be operational and ready but not yet enabled—it can but does not send nor receive data. This allows further stack setup as it delays receiving packets from the device. The network interface layer module is started and requested to register the network interface second. It registers itself as the packet supplier and consumer of the device. The inter–network layer module is started and requested to register the network interface at the nil last. The network interface is then started. The process is depicted in detail in the UML diagrams Section B.3.

4.1.5 Extending the networking stack

In this section a principle of integration of new modules into the networking stack is described. One of the goal was to develop an extendable networking stack and here is a brief manual.

Each new module has to be added as one of the *RD_SRVS* tasks in the make file `boot/arch/ia32/Makefile.inc` in the HelenOS source directory to be included in the boot image. Furthermore, the module service has to be defined in the header file `uspace/lib/libc/include/ipc/services.h` in the HelenOS source directory.

Most of the modules need to be registered by the *add_module()* function somewhere in the stack as well. This is due to the fact that the networking stack has to know the module in order to use it. Since then the module can be configured and used.

Network interface driver

A new network interface driver should implement the interface defined in the header file `netif/netif_module.h`.

The driver should be built with the `netif/netif.c` source file. The standalone driver is built with the `netif/netif_standalone.c` and `nil/nil_remote.c` source files while the bundle driver is built with the `netif/netif_nil_bundle.c` source file and the appropriate `nil/(module_name)/(module_name).c` source file. Please, see the `netif/lo/Makefile` as a reference of other needed source files.

In the *net* initialization function *net_initialize()* a module description has to be added by a function call similar to

```
add_module( NULL, & net_globals.modules, NAME, FILENAME, SERVICE, \
    0, connect_to_service ));
```

Network interface layer

A new network interface layer module should implement the interfaces defined in the header files `include/nil_interface.h` and `nil/nil_module.h` and process the messages defined in the `nil/nil_messages.h` header file.

The module should be built with a standalone module stub based on the source file `nil/nildummy/nildummy_module.c` changing the *NAME* definition and the *SERVICE* constants accordingly. Please, see the `nil/nildummy/Makefile` as a reference of other needed source files.

If the module is intended to function underneath the *arp* module, the protocol mapping needs to be defined in the *hardware_map()* function in the header file `include/protocol_map.h`. In the *net* initialization function *net_initialize()* a module description has to be added by a function call similar to

```
add_module( NULL, & net_globals.modules, NAME, FILENAME, SERVICE, \
    0, connect_to_service ));
```

Inter–network layer

A new inter–network layer module should implement the interface defined in the header file `include/il_interface.h` and process the messages defined in the header file `il/il_messages.h`.

The module should be built with a standalone module stub based on the source file `il/ip/ip_module.c` changing the *NAME* definition and the *SERVICE* constants accordingly. Please, see the `il/ip/Makefile` as a reference of other needed source files.

If the module is intended to function upon the *eth* module, the protocol mapping needs to be defined in the *protocol_[un]map()* and *lsap_[un]map()* functions in the `include/protocol_map.h` header file. The further networking stack integration depends on the module type.

General inter–network module In the *net* architecture specific initialization function *net_initialize_build()* a module description has to be added by commands similar to

```
task_id = spawn( FILENAME );
if( ! task_id ) return EINVAL;
ERROR_PROPAGATE( add_module( NULL, & net_globals.modules, NAME, \
    FILENAME, SERVICE, task_id, NAME_connect_module ));
```

for the modular networking, and

```
ERROR_PROPAGATE( REGISTER_ME( SERVICE, & phonehash ));
ERROR_PROPAGATE( add_module( NULL, & net_globals.modules, NAME, \
    FILENAME, SERVICE, task_get_id(), NAME_connect_module ));
ERROR_PROPAGATE( NAME_initialize( client_connection ));
```

for the monolithic build. Furthermore, the messages have to be forwarded in the monolithic build altering the *module_message()* function message switch like:

```
  switch( IPC_GET_TARGET( call )){
    case SERVICE:
```

```
      return NAME_message( callid, call, answer, answer_count );
...
}else if( IS_NAME_MESSAGE( call )){
  return NAME_message( callid, call, answer, answer_count );
```

and the *start_device()* function like:

```
switch( netif->il->service ){
  case SERVICE:
    ERROR_PROPAGATE( NAME_device_req( netif->il->phone, \
        netif->id, internet_service ));
```

ARP module A new ARP module should implement the interface defined in the header file `include/arp_interface.h` and process the messages defined in the `il/arp/arp_messages.h` header file. Multiple ARP modules are supported only for the modular networking architecture as the default *arp* is built and used in the bundle *net* module.

In the *ip* initialization function *ip_initialize()* a module description has to be added by a command similar to

```
ERROR_PROPAGATE( add_module( NULL, & ip_globals.modules, ARP_NAME, \
    ARP_FILENAME, SERVICE_ARP, arp_task_get_id(), \
    arp_connect_module ));
```

Transport layer

A new transport layer module should implement the interface defined in the header file `include/tl_interface.h` and process the messages defined in the header file `tl/tl_messages.h`.

The module should be built with a standalone module stub based on the source file `tl/udp/udp_module.c` changing the *NAME* definition and the *SERVICE* constants accordingly. Please, see the `tl/udp/Makefile` as a reference of other needed source files.

The module does not have to be added to the networking initialization, it can stand aside and can be started manually. If the module is to be integrated into the networking stack the following steps have to be taken. In the *net* architecture specific initialization function *net_initialize_build()* a module description has to be added by commands similar to

```
if( ! spawn( FILENAME )) return EINVAL;
```

for the modular networking, and

```
ERROR_PROPAGATE( REGISTER_ME( SERVICE, & phonehash ));
ERROR_PROPAGATE( NAME_initialize( client_connection ));
```

for the monolithic build. Furthermore, the messages have to be forwarded in the monolithic build altering the *module_message()* function message switch like:

```
  switch( IPC_GET_TARGET( call )){
    case SERVICE:
      return NAME_message( callid, call, answer, answer_count );
...
}else if( IS_NAME_MESSAGE( call )){
  return NAME_message( callid, call, answer, answer_count );
```

If the new protocol is intended to be published in the socket library a socket protocols mapping needs to be extended. A new static function to connect to the protocol module has to be implemented:

```
/** Returns the NAME module phone.
 *  Connects to the NAME module if necessary.
 *  @returns The NAME module phone.
 */
static int socket_get_NAME_phone( void );
```

and the mapping extended in the *socket()* function like:

```
switch( domain ){
  case PROTOCOL_FAMILY:
    switch( type ){
      case SOCK_TYPE:
        if( ! protocol ) protocol = IPPROTO_DEFAULT;
        switch( protocol ){
          case IPPROTO_NAME:
            phone = socket_get_NAME_phone();
            service = SERVICE;
            break;
```

4.1.6 *Qemu* network

In the common mode *Qemu* creates a simple network with a gateway and settles the guest system in. The network is 10.0.2.*, the gateway's address 10.0.2.2 and the guest system has 10.0.2.15. Therefore a static configuration is possible and no additional DHCP nor BOOTP implementations are necessary. On the other hand the guest system is behind a firewall. *Qemu* may be configured to forward some ports to the guest system and allows all outgoing traffic except ICMP and ARP protocols, so you can ping only the gateway.

4.1.7 *N.E.T.* user protocols

The protocol script is written in an XML file respecting the DTD file `protocol.dtd`. Each script should refer to the DTD file and comply with it. The whole process starting by a used protocol, connecting sequence, confirmation data, statuses and disconnecting sequence may be defined. The script has to be in the current directory while running the application. Script semantics is as follows

Identification Basic protocol settings such as displayed *name* and lower communication protocol (*lower* as `BuiltinTCP` or `BuiltinUDP`).

Connect A connecting sequence containing *send* and *receive* blocks and a target *state*. The application follows the sequence by sending and parsing the received data. If the sequence is successful the target state is achieved. Next connect sequence is tried instead if present.

Confirm A set of patterns used by the other host to confirm sent data. If any of the confirm patterns matches the sent data are confirmed.

State A set of uniquely numbered states (*number*) of the protocol. Each state may have a command sequence (*command*). There is an optional target state for a command the protocol achieves on success. The command may contain variable data marked as `%s` which the user can replace by any character string. The normal percent sign has to be written as `%%`.

All A set of commands like in the *state* block which may be used in any state.

Disconnect A disconnecting sequence similar to the connecting one. The connection is closed on success.

This can be of use for extending the networking stack with further protocols and applications in the future.

4.2 Running and testing

The networking stack is integrated into HelenOS to proof its operability. After starting either the provided boot images from the attached CD or built from source files in *Qemu*, the command line appears. The networking stack is started and initialized by running a command

```
# netstart
```

The networking stack is then started and configured network interfaces are enabled. The current configuration is printed out. Since that networking applications can be run using the command line as well.

4.2.1 Applications

A few networking applications are located in the `app/` directory. Common functions for parsing command line arguments and printing textual networking error messages are located in that directory as well.

The networking applications should be built with the *libsocket* library located in the `socket/libsocket.a` file. They can use functions and definitions from the `include/socket.h` header file which contains socket API and further includes:

- `include/byteorder.h` containing byte order manipulation,

- `include/in.h` containing IPv4 socket address structure,
- `include/in6.h` containing IPv6 socket address structure,
- `include/inet.h` containing socket address structure and parsing functions,
- `include/socket_codes.h` containing address and protocol families and socket types and option levels, and
- `include/socket_errno.h` containing socket and general error codes.

New applications can be added, for example into the `app/` directory. A new sub directory should be created with the source code and a simple Makefile

```
NAME = NAME_OF_THE_APPLICATION

NET_BASE = ../../
STRUCTURES = $(NET_BASE)structures/

include ../../../../../Makefile.config

## Sources
#

OUTPUT = $(NAME)
SOURCES = \
  $(NAME).c \
  ...

LIBS += ../../socket/libsocket.a

include $(NET_BASE)Makefile.module
```

The source code should include the `../../include/socket.h` header file. The next step is to tell networking stack to compile the module and HelenOS to include it into the boot image. An entry has to be added to the net `Makefile`

```
DIRS = \
  app/DIRECTORY/NAME_OF_THE_APPLICATION \
  ...
```

in the `boot/arch/ia32/Makefile.inc` file in the HelenOS source directory, it has to be added as one of the *RD_SRVS* tasks

```
RD_SRVS = \
$(USPACEDIR)/srv/net/app/DIRECTORY/NAME_OF_THE_APPLICATION \
...
```

The application can be run by the command then

```
# NAME_OF_THE_APPLICATION
```

4.2.2 Software prerequisites

The networking and TCP/IP stack is implemented for the *ia32* [7] architecture on top of HelenOS 0.4.1 (*Escalopino*) [8], the most current stable release of HelenOS. So far the only one operational network interface supported is in *Qemu* 0.10.2 and newer [9]. To run *Qemu* a script `contrib/conf/qemu.sh` for Linux or `contrib/conf/qemu.bat` for Windows in the HelenOS source directory can be used. The *qemu* and its libraries have to be installed and in the path. These scripts set all the necessary parameters with some ports redirected from the local host to the guest system. For testing purposes at least a low level communication application is recommended, *N.E.T.*, *netcat* etc.

In order to build HelenOS and the networking stack from sources a few tools are required:

- *binutils* in version 2.19.1 [10],
- *gcc–core* in version 4.3.3 [11],
- *gcc–objc* in version 4.3.3, and
- *gcc–g++* in version 4.3.3.

All these can be downloaded and installed as cross–compilers on Linux using a script `contrib/toolchain.sh` in the HelenOS source directory.

In addition *rats* [12], a static source code analyzer, and *Doxygen* [13], a documentation generator, were used. All development was tracked in the HelenOS *subversion* [14] repository.

[7] *IA32* is the most common (supported) architecture.
[8] HelenOS website: `<http://www.helenos.org/>`
[9] *Qemu* website: `<http://www.qemu.org/>`
[10] *binutils* website: `<http://www.gnu.org/software/binutils/>`
[11] *GCC* website: `<http://gcc.gnu.org/>`
[12] *RATS* website: `<http://www.fortify.com/security-resources/rats.jsp>`
[13] *Doxygen* website: `<http://www.stack.nl/~dimitri/doxygen/index.html>`
[14] *Subversion* website: `<http://subversion.tigris.org/>`

Chapter 5

Evaluation

In this section we compare the performance of the modular and the monolithic architectures which was one of the goals of this work.

In order to test all levels of the stack the topmost approach was chosen. It means a testing application using sockets to communicate with another one. The communication drifted down to a network interface and back up to the other application. The other application was the *echo* server just repeating back the received data. This simulates a real load on the networking stack and the localhost connection ensures testing of the whole communication process.

The default compile–time configuration for the *ia32* architecture was kept except that the *Debug build* was switched off. The networking stack was compiled as both modular and monolithic (Modular and Monolithic boot images).

Four versions of HelenOS were compiled for the testing purposes in fact. In addition to the configuration mentioned above, another two variants were compiled with one IP interface call cached (Modular-1 and Monolithic-1). The call was the *ip_packet_size_req()* function called by the *sendto()* function in the UDP module. This function call queries the IP module about the packet dimensions for the target destination. The UDP protocol is stateless and furthermore packets sent by one socket can have different destinations and can be routed via different network interfaces. Therefore the right packet dimensions should be agreed for each outgoing packet. This is the reason why this caching is sustainable only for demonstration purposes of the following tests.

HelenOS was always run anew in *Qemu* to avoid an interference with another system and to provide uniform conditions for each test. Therefore, the absolute results have to be taken with reserve as there is a significant slowdown when using an emulator.

The raw outcomes of the measurements are in the appendices Section A whereas average and aggregated values are discussed in this section.

5.1 Nettest2 – data transfer performance

The fist test measures the data transfer performance. The goal is to identify main factors influencing the data transfer performance.

A testing application *nettest2* was created performing the following functions

CR *sockets_create()* to create sockets

SR *sockets_sendto_recvfrom()* to send and receive datagrams of all sockets, each datagram is sent and a reply is received consequently so the next datagram is sent after the reply arrival

ST *sockets_sendto()* to send datagrams of all sockets, all datagrams are sent without receiving any replies

RF *sockets_recvfrom()* to receive reply datagrams of all sockets for all sent datagrams

SC *sockets_close()* to close sockets

There are command line arguments to specify the number of sockets (n), the number of datagrams (m) per socket and the datagram length (s). The application involves in total:

- one message to connect to the transport layer module,
- n IPC messages to create sockets,
- n IPC messages to close sockets, and
- $2 * n * m$ datagram transfers as depicted in the appendix Section B.4, each including
  - 26 IPC messages for Modular, 24 for Modular-1 and 12 for both Monolithic,
  - 4 times s–byte transfers of data,
  - 8 times 16 byte transfers of destination addresses for both Modular, 4 times for both Monolithic,
  - 2 times 12 byte transfers of IP pseudo headers for both Modular.

The total times taken of data transfers SR and ST+RF were measured.

The first test variant was set to use 1 socket and 100 datagrams each of 1024 bytes. The total amount of 100 kB of data was transferred both ways. The application was run twice by the command

```
# nettest2 -p 7 -t SOCK_DGRAM -n 1 -m 100 -s 1024 127.0.0.1
```

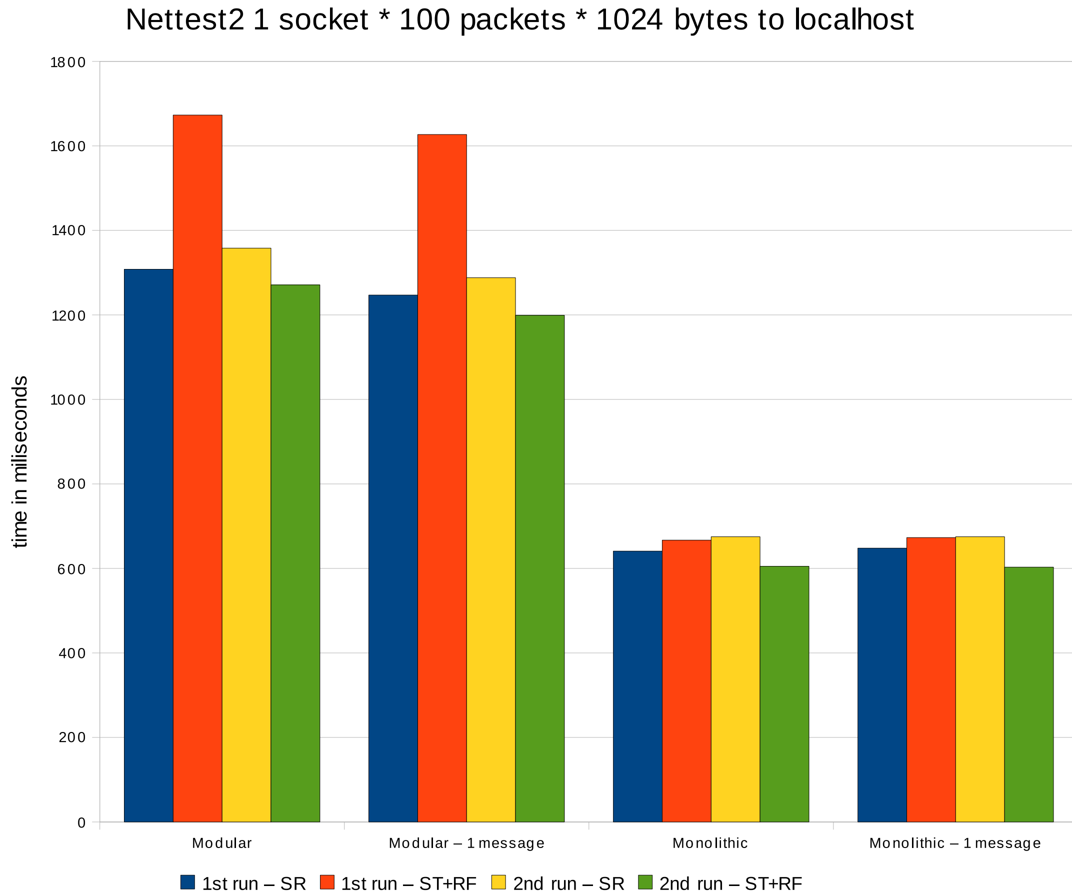


Figure 5.1: Nettest2

During the first run a sufficient number of packets was allocated and shared between all concerned stack modules. The second run used the allocated packets and the initial packet allocation and share overhead was avoided. There was no graphical output except the total time taken to measure the real processing time only.

The average values are depicted in the Figure 5.1 where we can see a few interesting phenomenons:

1. 1st runs of SR and ST+RF in the modular architecture differ a lot, by almost 30%. It is due to the fact that SR sends a datagram and waits for a reply. The datagram is received by the *echo* server and the reply is sent. There is only one packet allocated and used in the networking stack in this case. On the other hand ST+RF sends all datagrams before receiving any replies. Therefore the replies get buffered in the stack and the total number of 100 packets have to be created and shared between all 4 concerned modules (*udp*, *ip*, *nildummy*, *lo*). The difference is much smaller in the monolithic architecture as the packets are shared only with the network interface (*lo*). This overhead can be lowered either by decreasing the number of modules or optimization of the memory

sharing mechanism of HelenOS. The overhead is a singularity only when the networking stack allocates new packets and therefore the focus is on the packet reusability instead.

2. 2nd runs of SR take a bit more than 1st runs. This is probably caused by more often TLB page faults. There were allocated and shared many packets during ST+RF 1st run. These packets are mapped as memory pages in the networking stack modules' memory. Therefore a frequent reuse of packets can case TLB page faults more often. The 1st run has to be taken with reserve as it is a special case when there is only one packet in the networking stack. Further runs of the SR stabilize at a value close to the 2nd run.

3. 2nd runs of ST+RF take much less than 1st runs. It is caused by the packet reusability. All the needed packets were already allocated and shared during the 1st run. The initial packet allocation and share overhead was avoided.

4. 2nd runs of ST+RF take less time than SR. It is due to the fact that ST+RF does not have to wait for the reply and sends all datagrams whereas SR does wait.

5. Modular-1 takes a little less time than Modular, by 4.4%. This is caused by the cached IP interface call. The Modular involves the total of 2600 messages whereas the Modular-1 only 2400, which is 7.7% less.

6. The difference between Modular and Monolithic is significant. Monolithic took only 46.1% of time of the Modular. This is the major observation made in this test. The previous case gives us a clue that the number of IPC messages plays major role when talking about the performance. The Monolithic(-1) involves 1200 messages, only 46.2% of the Modular. Furthermore there is no copying of the IP pseudo header. There can be also a slowdown caused by the local networking stack used for both the client and the server. Locking mechanisms of global structures are used and received packets from the lower layer and sent packets from the upper can get serialized.

7. There is almost no difference between Monolithic and Monolithic-1. The cached IP interface call is not significant in this case as it is only a normal function call in the monolithic networking stack.

The second test variant of the test was set to use 1 socket and 100 datagrams each of only 33 bytes. Everything else remained unchanged. The influence of data transfer sizes was measured by this test.

The average values are compared to the previous results in the Figure 5.2 where we can see that the performance is almost the same as in the previous case. Therefore we can state that the data size does not influence the performance. However, thansfer of blocks of a size close to the MTU is more efficient.

The third test variant was set to use 10 socket and 10 datagrams each of 1024 bytes. Everything else remained unchanged. The influence of number of sockets was measured by this test.

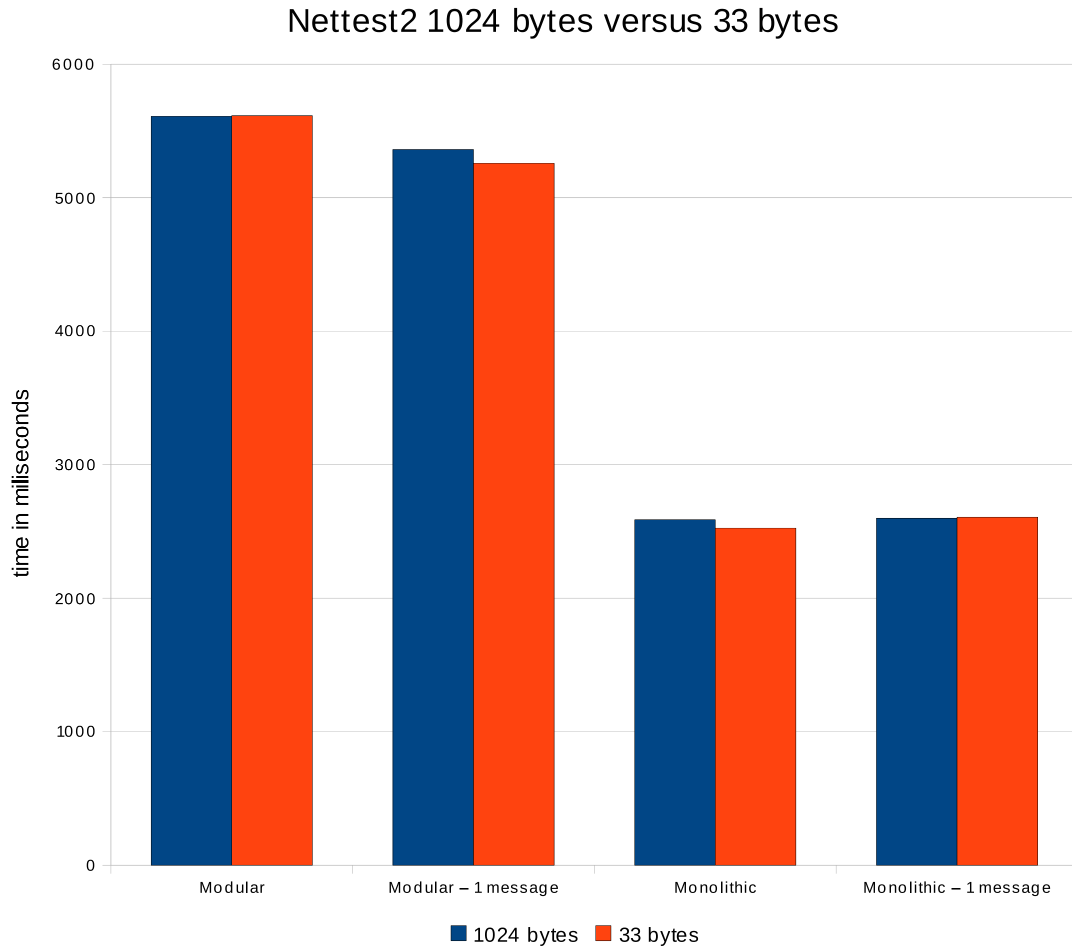


Figure 5.2: Nettest2, byte transfers

The average values are compared to the first set of results in the Figure 5.3 where we can see that the performance is slightly lower than in the first case. The number of active sockets influence the performance a bit but not significantly.

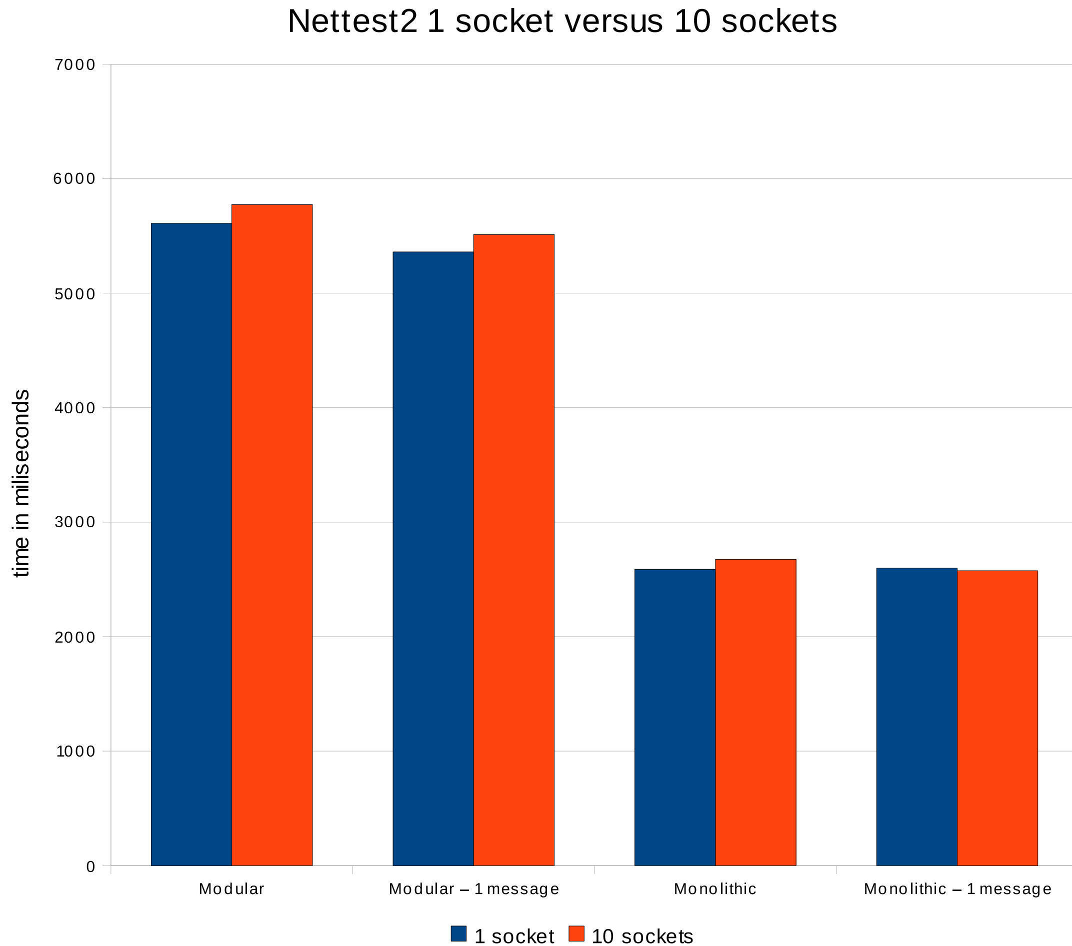


Figure 5.3: Nettest2, sockets

5.2 Nettest1 – the overall performance

This test measures the overall stack performance. The focus is on the socket and data transfer performance while increasing the number of sockets and packets in the networking stack.

The testing application *nettest1* creates, binds and closes sockets while sending and receiving testing data. It was originally designed as an integration test. It contains various schemes with one and many sockets transferring one and many datagrams. The schemes are combinations of the functions from the previous Section 5.1. The application runs the following sequences:

1. CR–CL, CR–SR–CL, CR–ST–RF–CL with one socket and one datagram

2. CR–SR–CL, CR–ST–RF–CL with one socket and m datagrams

3. CR–CL, CR–SR–CL, CR–ST–RF–CL with n sockets and one datagram

4. CR–SR–CL, CR–ST–RF–CL with n sockets and m datagrams

This progress increases the load of the networking stack and ensures its operability. There are command line arguments to specify the maximum number of sockets (n) and datagrams (m) and the datagram length (s). The application involves in total:

- one message to connect to the transport layer module,
- $5 * (n + 1)$ IPC messages to create sockets,
- $5 * (n + 1)$ IPC messages to close sockets, and
- $2*(n+m+1)+n*m$ datagram transfers as depicted in the appendix Section B.4, each including
  - 26 IPC messages for Modular, 24 for Modular-1 and 12 for both Monolithic,
  - 4 times s–byte transfers of data,
  - 8 times 16 byte transfers of destination addresses for both Modular, 4 times for both Monolithic,
  - 2 times 12 byte transfers of IP pseudo headers for both Modular.

The total time taken was measured.

The first test variant was set to use 10 sockets and 10 datagrams each of 1024 bytes. Therefore there will be a peak of 10 sockets and 100 buffered packets in the stack. The total amount of 142 kB of data is transferred both ways. The application was run twice by the command

```
# nettest1 -p 7 -t SOCK_DGRAM -n 10 -m 10 -s 1024 127.0.0.1
```

During the first run a sufficient number of packets is allocated and shared between all concerned stack modules. The second run uses the allocated packets and the initial packet allocation and share overhead is avoided. There was no graphical output except the total time taken to measure the real processing time only.

The average values are depicted in the Figure 5.4 where we can see that the difference between Modular and Monolithic is significant. The Monolithic took only 50% of the Modular. There can be many reasons as already mentioned. If we have a look at the Modular-1 it did a bit better. The Modular involved 3803 IPC messages whereas the Modular-1 involved only 3519 messages, which is 92.5%. The total time fell by 5.3% in this case. This complies with the *nettest2* results as the data transfer performance is major part of this test as well. Other parts of the test such as socket creating and closing involve only one IPC message each in both modular and monolithic variants. Therefore the result is the same as it was in the Section 5.1.

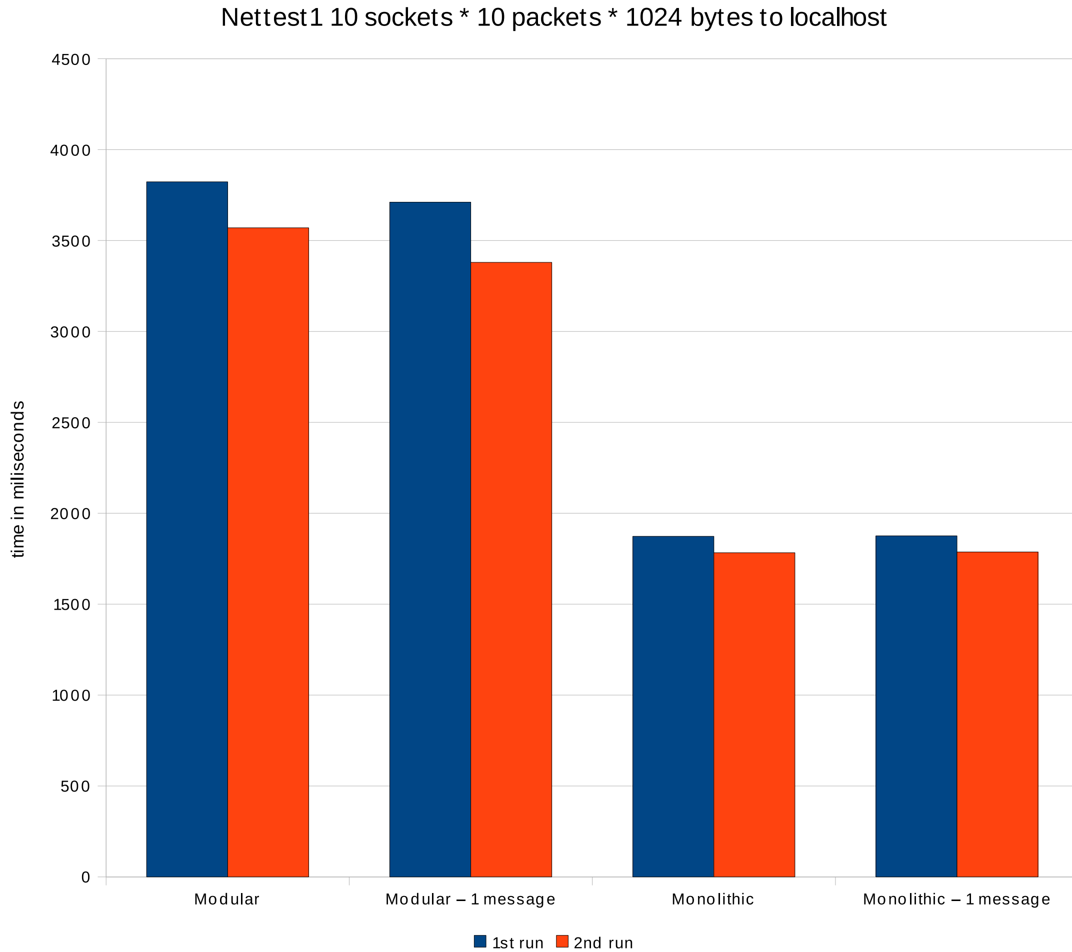


Figure 5.4: Nettest1

5.3 Ping – ICMP echo performance

The last test aimed at the responses of the localhost and a remote host. The localhost is reached via the loopback network interface whereas the 10.0.2.2 *Qemu* gateway via the emulated NE2000. With an assumption that *Qemu* does process the ICMP echo in almost the same time as the local networking stack, the difference between basic forwarding of a packet and transmitting a packet can be observed.

The testing application *ping* sends an echo request and waits for a reply. The time taken between the sent request and the received reply is printed out. The test was set to send 20 requests. The application was run by the commands

```
# ping -c 20 127.0.0.1
```

and

```
# ping -c 20 10.0.2.2
```

respectively.

The first requests were discarded because they were influenced by a packet allocation and sharing. Furthermore, in the case of *Qemu* the reply did not arrive at all as the ARP mapping did not yet exist and an ARP request was sent instead. The cached IP interface call does not play any role in this test so the results were put together for the Modular and Monolithic variants. The average values are depicted in the Figure 5.5.

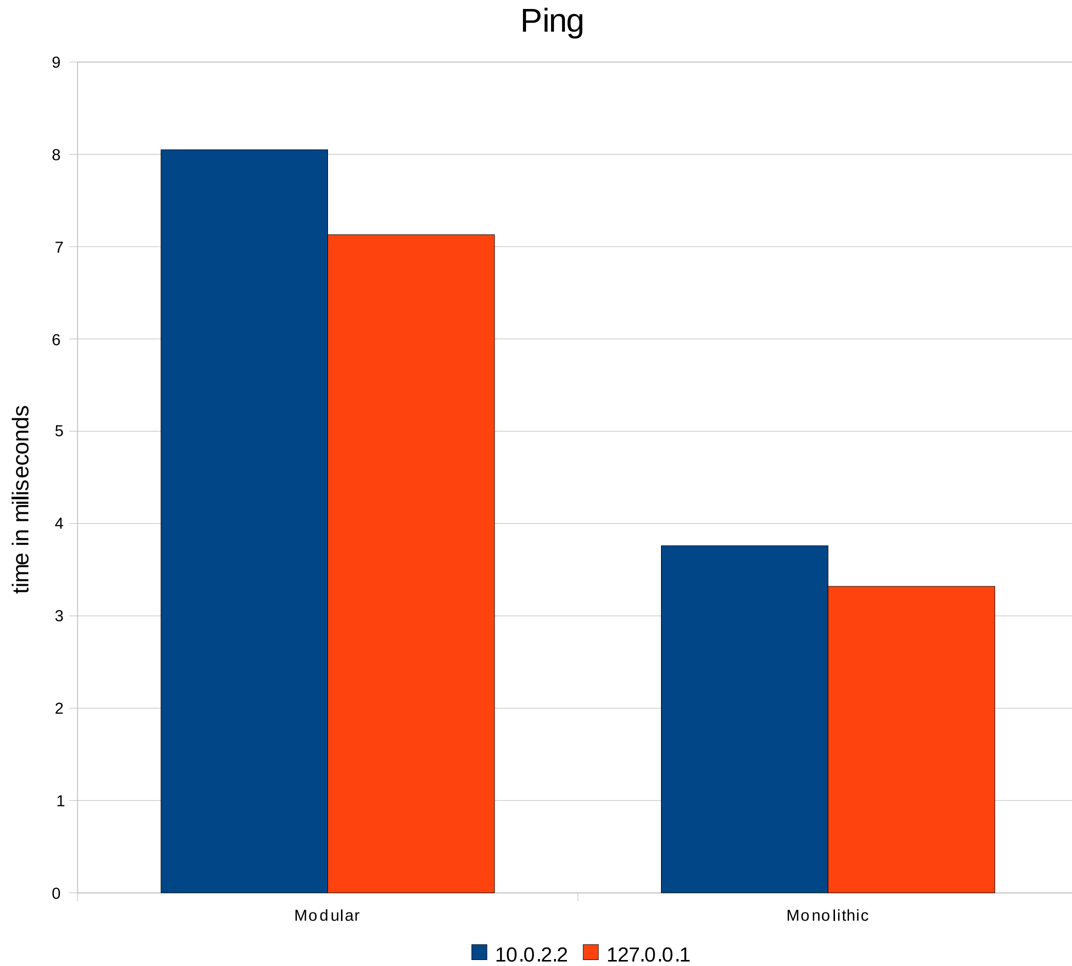


Figure 5.5: Ping

The results show that the emulated NE2000 did a bit worse which was expected. Furthermore, the difference in the performance of the architectures persists.

5.4 Conclusion

According to the test results we can point out some observations. The most obvious is the overall difference between the modular and monolithic architecture. Despite

the results the modular architecture could be further optimized and its advantages could overbalance the performance loss.

There is a significant overhead when new packets are created and shared between networking stack modules. This overhead could be lowered only by optimizing the HelenOS memory sharing.

On the other hand the slowdown caused by IPC messages in general is more important. It influences the overall and long–run performance of the networking stack. In order to keep the modular design the total number of modules cannot be lowered. Optimization and caching of IPC messages should be performed instead. There could be introduced faster IPC calls for frequently used messages using less arguments or some sorts of piggybacking. Arguments while passing packets could be also transferred in the packets themselves as are the addresses now. The locking concept in the networking stack modules can be also revised if there could be more shared accesses and less exclusive ones.

Interesting is the fact that the packet size does not influence the performance. Although the same data would get transferred in more packets the data copying into packets itself does not slow down the datagram transfers.

Chapter 6

Other architectures

In this section we describe other operating systems' networking stacks. We write about the general concept of the stacks, the authors' motivation and goals and some of the main benefits and features. The list is focused on the main interesting distributions.

6.1 BSD

Berkeley Software Distribution was the first operating system implementing the TCP/IP Suite. Their first implementation appeared in 1983 in the 4.2 BSD after a close cooperation with ARPA. The focus was on the protocol suite itself. The 4.4 BSD networking stack is often considered as the reference implementation [11]. It allowed the TCP/IP Suite to spread wide. The BSD networking stack introduced socket API which became a de facto standard as well.

The BSD stack attempts to be the fastest networking stack. It was the first stack breaking 1Mpps, routing of one million packets per second [14].

The networking stack is implemented in the kernel as one subsystem. Therefore interface calls are just normal function calls which increases performance as in the case of our monolithic configuration.

The stack uses *mbuf* structure to hold packets. The *mbuf* is a fixed size block and larger packets are chains of such blocks. The fixed size is a requirement of a SLAB allocator. Although the buffers are not reused as our packets, the SLAB allocator does a very similar thing. It reserves the buffers in the memory and allocates them as needed. The allocation overhead is there for each new packet, however, it is much lower than our memory sharing. On the other hand our packet reuse takes just one IPC message and uses no dynamic allocation.

The security of the BSD networking stack tends to be also very high [3]. The authors of the stack discovered many vulnerabilities of protocols before they were abused. They try to use pseudo–random number generators where possible which led to resistance against some blind insertion attacks targeted on sequence numbering. The counters, session identifiers and timestamps are some of the examples.

Furthermore, the stack does not implement all features of RFCs as some were

considered vulnerable or needless. The stack ignores IP options, and some suspicious ICMP errors for example [15].

BSD uses its own network interface drivers and *NDISwrapper*, which is a quite controversial project. NDIS compatible drivers are included and used by the BSD stack. These drivers are mostly third party drivers developed for Microsoft Windows NDIS API. Therefore the wrapper introduces third party and often closed code right into the kernel mode. This approach emerged from the fact that there are many network interfaces and the manufacturers often support only the NDIS API.

There is one specialty with socket identifiers. Sockets identifiers in BSD refer to files so the socket API and even the file API can be used to send and receive data. The sockets are propagated to applications as regular files whereas the system treats them in a different way. This allows passing sockets to some reading or writing functions as file descriptors. On the other hand it needs deeper system integration of the socket API. Our networking stack stands still apart from other subsystems of HelenOS.

6.2 Linux

The Linux networking stack [2] is very similar to the BSD networking stack, the whole stack is in the kernel, uses BSD socket API, sockets as special files, a packet buffer called *sk_buff*, its own drivers and *NDISwrapper*. However, it is not a clone of the BSD networking stack, it has just the same philosophy.

The stack is in fact a compound of many kernel modules. Nevertheless, these modules are compiled or loaded into the kernel and run in the kernel mode. The modules register their interface functions as pointers and use global data structures which reside in the kernel memory. This is for performance reasons again.

The programming security reasons stay a bit aside even though the Linux is an open source project and has thousands of developers. It is, therefore, hard to revise all the source code running in the kernel mode.

Linux kernel contains a *netfilter* which filters the network traffic according to configured rules. It uses hooks in the networking stack to be informed about packets at certain times of the processing. This hooks are implemented as function pointers as well to preserve the performance.

The Linux kernel itself is influenced by the networking stack. One example is received frames retrieval. There are four possibilities drivers can use. Nevertheless, the drivers are not forced to use the most efficient. An interrupt handler can be used to transfer the frames into the driver's receive queue while blocking any other process on the CPU as the interrupts are disabled. Another option is to use bottom half and top half interrupt handlers where only the necessary part of the interrupt handling is treated by the top half. The bottom half is deferred and runs as a normal process then. The bottom halves were internally changed to the third option, softirqs. The softirq runs as a normal thread with interrupts enabled, however, the kernel does not allow another softirq to run on the CPU before the previous finishes. This decreased the overhead as the kernel was altered a bit. The last option is polling where the

driver asks frequently the network interface to provide received frames. This is very useful when the load is too high that there are so many interrupts that the driver can barely get to the CPU to retrieve some of the received frames and the newly received ones are discarded.

In comparison to HelenOS, only the IRQ recognition command sequence is run in the kernel interrupt handler with interrupts disabled. The driver is notified by an IPC message and the callback function does the processing. Although it is similar to the bottom half, the standard IPC mechanism is used.

6.3 Windows

Microsoft Windows are the most used operating systems. The networking stack was integrated in early 1990s and used the *NetBIOS* interface. A newer interface *Winsock* was added in 1993, and its newer version 2.2 comes from 1996. The *Winsock* interface is quite similar to the BSD socket interface and it supports both synchronous and asynchronous socket function calls.

The first version of the stack used some parts of the BSD implementation. However, the stack was completely rewritten for Windows 2000 [13]. The Windows 2000 was an “Internet–ready” release of Windows. It provided support for multiple network interfaces, IP routing, default gateways, VPNs, firewall hooks and other technologies. The stack was in the kernel and accessible by system calls as well. At the bottom of the stack there was a packet scheduler, a central module for queueing and scheduling packets on top of NDIS drivers. This gave another layer of abstraction and shifted some common functionality from drivers to the stack. A piquancy was the emulation of wireless network interfaces as normal wired once.

Nevertheless, there is already a newer version of the stack since Windows Vista [5]. It is called the “Next Generation TCP/IP Stack”. The stack is still an integral part of the operating system and implements much more RFCs than the previous stack. The stack is internally more modular and dynamic. Its configuration can be changed even when running.

This release removed the packet scheduler and the wired emulation of wireless network interfaces. The overall support for wireless network interfaces is much better. This release implements native IPv6 and changes the network layer into a dual IP layer with both IPv4 and IPv6 available. There are also security improvements in easier IPsec configuration and a new Windows Filtering Platform which allows other applications to use the filtering engine integrated into the networking stack.

6.4 Minix

Minix is probably the most famous microkernel because of its author Andrew Tanenbaum [27]. The system is under development since 1980s.

Although Minix is a microkernel operating system, the networking stack is in one large module using BSD sockets API. Currently, there is a plan to break its

networking module into separate server modules [1]. Their results could be interesting in comparison to this work.

Minix does not support many network interfaces, however, the drivers are simple and functional. One of them is ported in our stack. The drivers support vectored I/O which allows packets to be chains of memory blocks. These block are not shared but copied from the networking stack to the drivers' address spaces by system calls. There is not a single packet buffer structure.

Other microkernel operating systems tend to use the monolithic networking module as well (EROS, Mach, JariOS, QNX Neutrino RTOS . . .).

[1] A project to split the Minix networking stack into number of drivers and servers (IP, UDP, TCP...) was put up in October 2009.

Chapter 7

Conclusion

A functional networking stack is a complex subsystem as it spans over many levels of abstraction. Nevertheless, there is just a simple interface on top of it. A lot of work is to be done under the hood to make the stack operational.

In this work we analyzed a networking stack in general. Modular and monolithic architectures were introduced and compared. The modular architecture can be viewed as a service oriented architecture whereas the monolithic as a standalone subsystem. The advantages and disadvantages of both architectures were discussed.

The main focus of the work was set to a modular networking stack for a microkernel operating system. Having the modern concept in mind the author of this thesis evaluated and put together the age–proved protocols to create a new implementation.

The stack was also designed to support both modular and monolithic build which allowed us to compare the modular stack to its monolithic equivalent. Some strengths and weaknesses of the modular architecture were pointed out. One of the most mentioned advantages of the monolithic architecture is the performance which was also backed up by test results. The main performance factors were analyzed and some optimizations were proposed which could reduce the inequality of performance.

The idea of a strictly modular networking stack appeared independently in the Minix community a short time ago which supports our first intention as there is interest in this topic. It might be a valuable comparison to this work if finished.

The presented master thesis fulfils its goals laid out in the introduction. The networking stack design was evaluated and revised.A functional implementation in HelenOS was successfully achieved which made the main contribution of the work possible, the comparison of the modular and monolithic architectures. Although the results are not too supportive, we tried to give reasons to overbalance the performance of the monolithic architecture in favour of the modular design. Furthermore, the implemented networking stack is going to become an integral part of HelenOS.

Future work

The implemented networking stack was just a starting point for a long–term run. The networking stack features and capabilities should be improved and extended in

successive projects. One field are the optimizations of IPC messages. The networking stack implementation for a productive system should reduce the overhead and increase its effectiveness.

Some often used features of the stack should be the goals of the first development. The the TCP protocol is not finished yet and some enhancement and optimization RFCs could be also added. An IP fragments reassembly could be based on a TCP packet buffering and timeout fibrils whereas the dynamic IP configuration involves new service modules running aside. The DHCP protocol uses UDP sockets for example.

A support for network interfaces is very limited now. Therefore a development of new drivers is highly recommended as it will help the networking stack to spread. The designed driver architecture is, hopefully, flexible enough to allow easy development and usage of additional drivers.

Another field for extensive work is the overall security. It might be useful and desirable to implement packet filtering and some sort of restrictions for ports and module extensions.

And last but not least, porting of many networking applications might follow.

Chapter 8

Terms and abbreviations

Here is a list of main networking terms and abbreviations. The abbreviations are divided into organization names, protocols and others.

Terms

Host A single communicating or a network connected computer.

Network interface A piece of hardware providing networking communication, informally a network card.

Networking stack An actual implementation of cooperative networking protocols in order to enable inter–computer communication.

Octet One byte in the networking terminology.

Packet A block of formatted data carried in a network.

Socket A virtual network connection between two hosts.

Sockets An API in the context of a networking stack, usually a BSD sockets interface is meant.

Organizations

ARPA Advanced Research Projects Agency, the American institution ARPA is called DARPA (D as Defense) now.

IANA Internet Assigned Numbers Authority, the central authority overseeing IP addresses, protocol numbers and parameters allocation as well as root zone for the DNS.

IEEE Institute of Electrical and Electronics Engineers, a worldwide professional organization focusing on electrical, electronics and information technology.

IETF Internet Engineering Task Force, an open standards organization with focus on developing and promoting Internet standards.

Protocols

ARP Address Resolution Protocol, a protocol used for mapping logical network addresses to underlying network addresses.

BOOTP Bootstrap Protocol, a protocol for an automatic configuration of a host in a computer network. The host uses this protocol only at its networking startup.

DHCP Dynamic Host Configuration Protocol, a protocol for an automatic dynamic configuration of a host in a computer network.

ICMP Internet Control Message Protocol, an error notification and support protocol used in the TCP/IP Suite, serves to notify the other party about errors and transfer network information.

IP Internet Protocol, a key inter–networking protocol in the TCP/IP Suite, it simulates a connectionless, unreliable and best–effort packet delivery network.

TCP Transmission Control Protocol, a connection oriented and reliable transport protocol upon IP.

UDP User Datagram Protocol, a connectionless, unreliable and best–effort transport protocol upon IP.

Other abbreviations

ARPANET ARPA Network, the original name of the first computer network.

BSD Berkeley Software Distribution, a Unix operating system developed at the University of Colorado. The first operating system implementing the TCP/IP Suite.

CRC Cyclic Redundancy Check, a hash function computing a checksum.

DNS Domain Name System, a mechanism of translation textual addresses to IP addresses.

IPsec Internet protocol security, a universal protocol suite for securing IP protocol packets. It provides authentication, encryption and cryptographic keys negotiation.

ISO/OSI International Organization for Standardization / Open System Interconnection, a key networking reference model with seven layers of decomposition—Application, Presentation, Session, Transport, Network, Data Link and Physical—but without concrete protocol specification.

MAC address Media Access Control address, a physical address of a network interface. This address is used to identify the source and the destination network interfaces of frames in the data link networking stack layer.

MTU Maximum Transmission Unit, the maximum size of data which can be transferred at once. The MTU can be limited by a present as well as by a remote network interface.

NDIS Network Driver Interface Specification, an API of network interfaces developed by Microsoft and 3Com Corporation.

NetBIOS Network Basic Input/Output System, an original networking API used in early Microsoft Windows.

RFC Request for comments, an IETF memorandum on Internet systems and standards

TCP/IP The TCP/IP Suite is a networking architecture with four layers—Application, Transport, Inter–network and Network Interface—and concrete protocol specifications.

Bibliography

[1] Appendix A, The Tanenbaum-Torvalds Debate in *Open Sources: Voices from the Open Source Revolution* [on–line]. O'Reilly Media, January 1999, First edition. [cited November 20 2009]. ISBN 10: 1-56592-582-3. Available on the World Wide Web
`<http://www.oreilly.com/catalog/opensources/book/appa.html>`

[2] BENVENUTI, C *Understanding Linux Network Internals*. O'Reilly Media, December 29 2005. ISBN 13: 978-0-596-00255-8.

[3] BIANCUZZI, F. *Security Focus*. [on–line community]. Document version of 2005-12-10. [cited December 5 2009]. OpenBSD's network stack. Available on the World Wide Web
`<http://www.securityfocus.com/columnists/361/1>`.

[4] BRADEN, R.T.; BORMAN, D.A.; PATRIDGE, C. *RFC 1071: Computing the Internet checksum* [on–line]. September 1 1988. [cited November 20 2009] Available on–line
`<ftp://ftp.isi.edu/in-notes/rfc1071.txt>`.

[5] DAVIES, J *TCP/IP Fundamentals for Microsoft Windows*. [on–line]. Microsoft Corporation, Document version of 2008-02-06. [cited December 7 2009]. Available on the World Wide Web
`<http://www.microsoft.com/downloads/details.aspx`
`?displaylang=en&FamilyID=c76296fd-61c9-4079-a0bb-582bca4a846f>`.

[6] DĚCKÝ, M. Component–based General–purpose Operating System. In *WDS'07 Proceedings of Contributed Papers, Part I*. Matfyzpress, 2007, 58—63. [cited November 14 2009]. ISBN 13: 978-80-7378-023-4. Available on the World Wide Web:
`<http://www.mff.cuni.cz/veda/konference/wds/contents/pdf07/`
`WDS07_110_i2_Decky.pdf>`.

[7] *DP8390D/NS32490D NIC Network Interface Controller*. National Semiconductor Corporation, July 1995. [cited November 17 2009]. Available on the World Wide Web:
`<http://www.national.com/ds.cgi/DP/DP8390D.pdf>`.

[8] *HelenOS 0.2.0 design documentation* [on–line]. June 18 2006. [cited November 14 2009]. Available on the World Wide Web:
`<http://www.helenos.org/doc/design.pdf>`.

[9] LIEDTKE, J. On Microkernel Construction. In *Proceedings of the 15th ACM Symposium on Operating System Principles (SOSP-15)*. Copper Mountain Resort, Colorado, December 1995. [cited November 25 2009]. Available on the World Wide Web
`<http://i30www.ira.uka.de/research/documents/l4ka/1995/ukernel-construction.pdf>`

[10] MALLORY, T.; KULLBERG, A. *RFC 1141: Incremental updating of the Internet checksum* [on–line]. January 1 1990. [cited November 20 2009] Available on–line
`<ftp://ftp.isi.edu/in-notes/rfc1141.txt>`.

[11] MCKUSICK, M.K. et al. *The Design and Implementation of the 4.4BSD Operating System*. Addison Wesley Professional, May 10 1996, Second edition. ISBN 10: 0-201-54979-9.

[12] METCALFE, R.M.; BOGGS, D.R Ethernet: distributed packet switching for local computer networks. In *Communications of the ACM*. ACM, July 1976, vol. 19, no. 7, 395–404. ISSN: 0001-0782.

[13] Microsoft Corporation. *Microsoft TechNet*. [on–line knowledge database]. Microsoft Windows 2000 TCP/IP Implementation Details. [cited December 7 2009]. Available on the World Wide Web
`<http://technet.microsoft.com/en-us/library/bb726981.aspx>`.

[14] OPPERMANN, A. *FreeBSD 5 Network Enhancements*. [on–line] SUCON 04, September 3 2004. [cited December 4 2009]. Available on the World Wide Web
`<http://people.freebsd.org/~andre/FreeBSD-5.3-Networking.pdf>`.

[15] OPPERMANN, A. *New Networking Features in FreeBSD 6*. [on–line] EuroBSDCon 05. [cited December 5 2009]. Available on the World Wide Web
`<http://people.freebsd.org/~andre/New%20Networking%20Features%20in%20FreeBSD%206.pdf>`.

[16] PARZIALE, L. et al. *TCP/IP Tutorial and Technical Overview* [on–line]. International Business Machines Corporation Redbooks, December 19 2006, Eight edition. [cited December 27 2008]. ISBN 10: 0738494682. Available on the World Wide Web
`<http://www.redbooks.ibm.com/redbooks/pdfs/gg243376.pdf>`.

[17] POSTEL, J. *RFC 768: User Datagram Protocol* [on–line]. August 28 1980. [cited November 20 2009] Available on–line
`<ftp://ftp.isi.edu/in-notes/rfc768.txt>`.

[18] POSTEL, J. *RFC 791: Internet Protocol* [on–line]. September 1 1981. [cited November 20 2009] Available on–line
<ftp://ftp.isi.edu/in-notes/rfc791.txt>.

[19] POSTEL, J. *RFC 792: Internet Control Message Protocol* [on–line]. September 1 1981. [cited November 20 2009] Available on–line
<ftp://ftp.isi.edu/in-notes/rfc792.txt>.

[20] POSTEL, J. *RFC 793: Transmission Control Protocol* [on–line]. September 1 1981. [cited November 20 2009] Available on–line
<ftp://ftp.isi.edu/in-notes/rfc793.txt>.

[21] PLUMMER, D.C. *RFC 826: Ethernet Address Resolution Protocol: Or converting network protocol addresses to 48.bit Ethernet address for transmission on Ethernet hardware* [on–line]. November 1 1982. [cited November 20 2009] Available on–line
<ftp://ftp.isi.edu/in-notes/rfc826.txt>.

[22] RIJSINGHANI, A. *RFC 1624: Computation of the Internet Checksum via Incremental Update* [on–line]. May 1994. [cited November 20 2009] Available on–line
<ftp://ftp.isi.edu/in-notes/rfc1624.txt>.

[23] RUBINI, A.; CORBET, J. *Linux Device Drivers* [on–line]. O'Reilly, June 2001, Second edition. [cited July 26 2008]. ISBN 10: 0-596-00008-1. Available on the World Wide Web:
<http://www.xml.com/ldd/chapter/book/index.html>.

[24] SRIVASTAVA, V.; MOTANI, M. Cross–layer design: a survey and the road ahead. In *Communications Magazine*. IEEE, December 2005, vol. 12, no. 1, 112–119.

[25] SUTTON, C. *UCLA*. [on–line magazine]. Document version of 2004-09-02. [cited December 5 2009]. Internet Began 35 Years Ago at UCLA with First Message Ever Sent Between Two Computers. Available on the World Wide Web
<http://www.engineer.ucla.edu/stories/2004/Internet35.htm>.

[26] TANENBAUM, A.S. *Computer Networks*. Prentice Hall, 2002. [cited November 14 2009]. ISBN 10: 0130661023. Available on the World Wide Web
<http://books.google.cz/books?id=Pd-z64SJRBAC>.

[27] TANENBAUM, A.S.; WOODHULL, A.S. *Operating Systems Design and Implementation, 3/E*. Prentice Hall, April 1 2006, Third edition. ISBN 13: 9780131429383.

[28] WINSTON, I. *RFC 948: Two methods for the transmission of IP datagrams over IEEE 802.3 networks* [on–line]. June 1 1985. [cited November 20 2009] Available on–line
<ftp://ftp.isi.edu/in-notes/rfc948.txt>.

[29] WONG, C.T. *Minix Networking Documentation*. [on–line]. Document version of 2003-12-04. [cited December 6 2009]. Available on the World Wide Web
<http://www.nyx.net/∼ctwong/minix/Minix_startpage.htm>.

[30] ZAGHAL, R.Y.; KHAN, J.I. *EFSM/SDL modeling of the original TCP standard (RFC793) and the Congestion Control Mechanism of TCP Reno* [on–line]. Kent State University, July 22 2005. [cited November 20 2009] Available on the World Wide Web:
<http://www.medianet.kent.edu/techreports/
TR2005-07-22-tcp-EFSM.pdf>.

[31] ZIMMERMANN, H. OSI Reference Model – The IS0 Model of Architecture for Open Systems Interconnection. In *Transactions on Communications*. IEEE, April 1980, vol. 28, no. 4, 425–432. [cited November 14 2009]. Available on the World Wide Web
<http://www.comsoc.org/livepubs/50_journals/pdf/
RightsManagement_eid=136833.pdf>.

Appendix A

Test results

Nettest1 10 sockets * 10 packets * 1024 bytes to localhost

Run	Modular				Modular - 1 message			
1st	3838	3816	3815	$\simeq 3823$	3759	3722	3653	$\simeq 3711$
2nd	3583	3557	3571	$\simeq 3570$	3379	3415	3346	$\simeq 3380$
Run	**Monolithic**				**Monolithic - 1 message**			
1st	1878	1869	1871	$\simeq 1873$	1878	1875	1876	$\simeq 1876$
2nd	1791	1781	1777	$\simeq 1783$	1800	1785	1777	$\simeq 1787$

(times in miliseconds)

Nettest1 10 sockets * 10 packets * 38 bytes to localhost

Run	Modular				Modular - 1 message			
1st	3781	3852	3885	$\simeq 3839$	3649	3619	3626	$\simeq 3631$
2nd	3493	3583	3585	$\simeq 3554$	3358	3324	3325	$\simeq 3336$
Run	**Monolithic**				**Monolithic - 1 message**			
1st	1850	1844	1839	$\simeq 1844$	1839	1837	1860	$\simeq 1845$
2nd	1755	1748	1746	$\simeq 1750$	1741	1751	1758	$\simeq 1750$

(times in miliseconds)

Nettest2 1 socket * 100 packets * 1024 bytes to localhost

Run	Mode	Modular				Modular - 1 message			
1st	ST + RF	1303	1314	1306	$\simeq 1308$	1254	1241	1247	$\simeq 1247$
1st	SR	1662	1686	1671	$\simeq 1673$	1638	1617	1626	$\simeq 1627$
2nd	ST + RF	1358	1361	1354	$\simeq 1358$	1300	1284	1281	$\simeq 1288$
2nd	SR	1269	1268	1276	$\simeq 1271$	1215	1188	1193	$\simeq 1199$

Run	Mode	Monolithic				Monolithic - 1 message			
1st	ST + RF	655	634	633	$\simeq 641$	639	675	630	$\simeq 648$
1st	SR	671	665	664	$\simeq 667$	668	693	657	$\simeq 673$
2nd	ST + RF	680	675	669	$\simeq 675$	672	682	670	$\simeq 675$
2nd	SR	602	608	604	$\simeq 605$	598	606	605	$\simeq 603$

(times in miliseconds)

Nettest2 1 socket * 100 packets * 33 bytes to localhost

Run	Mode	Modular				Modular - 1 message			
1st	ST + RF	1298	1323	1301	$\simeq 1307$	1229	1235	1214	$\simeq 1226$
1st	SR	1667	1705	1677	$\simeq 1683$	1596	1602	1581	$\simeq 1593$
2nd	ST + RF	1347	1372	1349	$\simeq 1356$	1271	1280	1256	$\simeq 1269$
2nd	SR	1261	1278	1265	$\simeq 1268$	1168	1180	1163	$\simeq 1170$

Run	Mode	Monolithic				Monolithic - 1 message			
1st	ST + RF	620	622	625	$\simeq 622$	660	643	631	$\simeq 645$
1st	SR	649	660	652	$\simeq 654$	759	664	659	$\simeq 694$
2nd	ST + RF	663	663	664	$\simeq 663$	684	667	661	$\simeq 671$
2nd	SR	582	584	591	$\simeq 586$	608	591	592	$\simeq 597$

(times in miliseconds)

Nettest2 10 sockets * 10 packets * 1024 bytes to localhost

Run	Mode	Modular				Modular - 1 message			
1st	ST + RF	1359	1334	1334	$\simeq 1342$	1268	1278	1293	$\simeq 1280$
1st	SR	1746	1714	1715	$\simeq 1715$	1644	1655	1674	$\simeq 1658$
2nd	ST + RF	1412	1391	1387	$\simeq 1397$	1325	1333	1337	$\simeq 1332$
2nd	SR	1330	1324	1307	$\simeq 1320$	1230	1246	1251	$\simeq 1242$

Run	Mode	Monolithic				Monolithic - 1 message			
1st	ST + RF	691	649	658	$\simeq 666$	658	646	695	$\simeq 666$
1st	SR	705	684	695	$\simeq 695$	700	687	711	$\simeq 600$
2nd	ST + RF	694	697	706	$\simeq 699$	692	698	699	$\simeq 696$
2nd	SR	608	613	624	$\simeq 615$	609	619	615	$\simeq 614$

(times in miliseconds)

Ping 20 times localhost

Modular	28	7	9	6	7	7	7	9	7	7	
	6	7	9	7	7	7	8	9	8	7	$\simeq 7.05$
Modular - 1 message	28	7	8	7	8	10	7	7	7	7	
	10	7	7	8	7	9	7	7	7	8	$\simeq 7.21$
Monolithic	18	4	4	4	4	4	4	4	4	3	
	4	3	4	4	4	3	3	4	3	4	$\simeq 3.53$
Monolithic - 1 message	16	4	3	3	3	3	4	3	3	3	
	4	3	3	4	4	3	3	3	3	3	$\simeq 3.11$

(times in miliseconds)

Ping 20 times 10.0.2.2

Modular	-	11	8	8	8	9	7	8	7	8	
	9	9	9	8	8	8	8	11	9	8	$\simeq 8.05$
Modular - 1 message	-	11	9	9	8	8	8	8	8	8	
	11	8	8	8	8	7	8	10	8	8	$\simeq 8.05$
Monolithic	-	4	4	3	6	3	4	4	3	4	
	4	5	3	4	4	3	4	3	4	4	$\simeq 3.63$
Monolithic - 1 message	-	6	3	4	4	4	4	4	4	4	
	4	5	4	4	4	4	4	4	4	4	$\simeq 3.89$

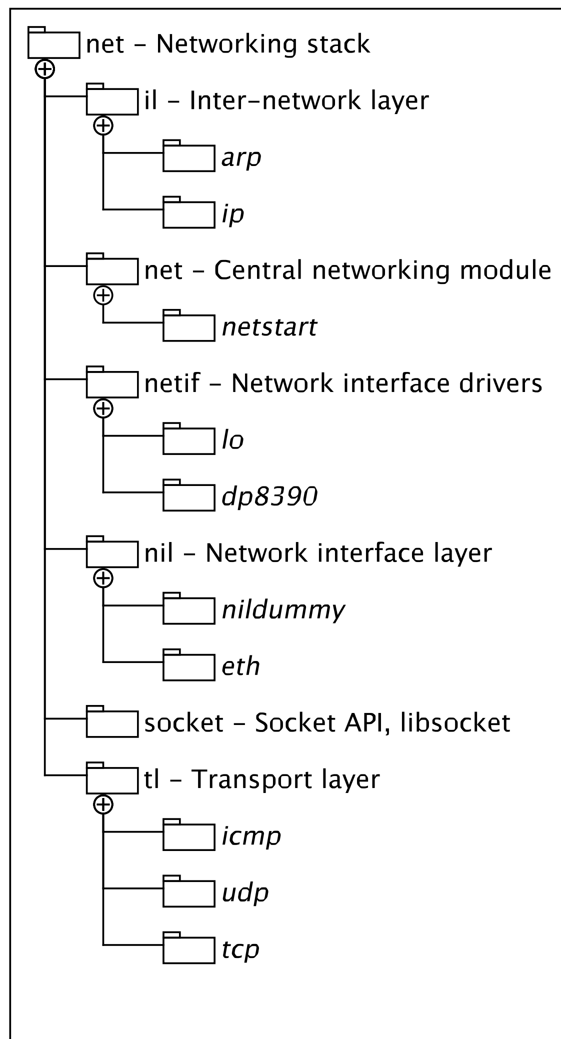
(times in miliseconds)

Appendix B

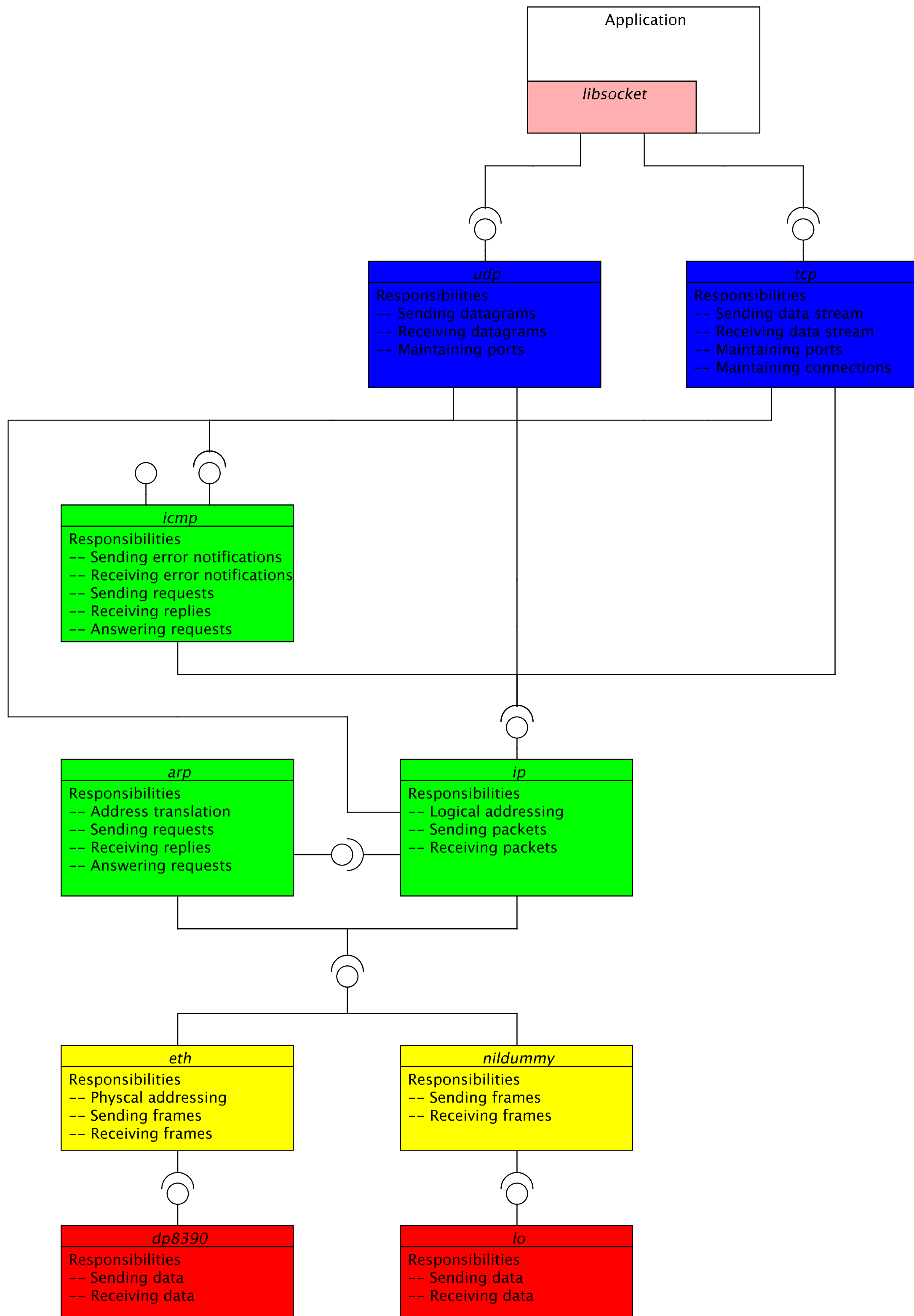
UML diagrams

B.1 Networking stack overview

Networking stack directories

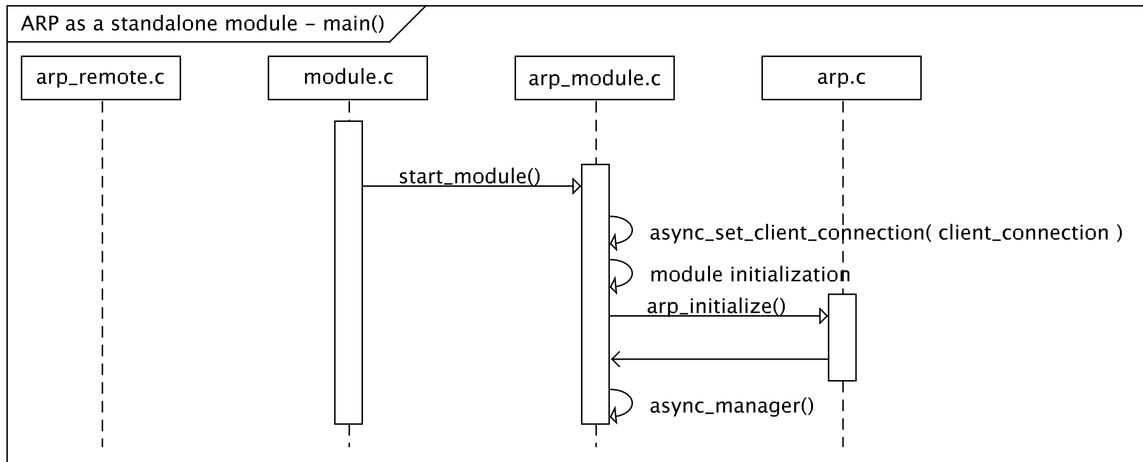


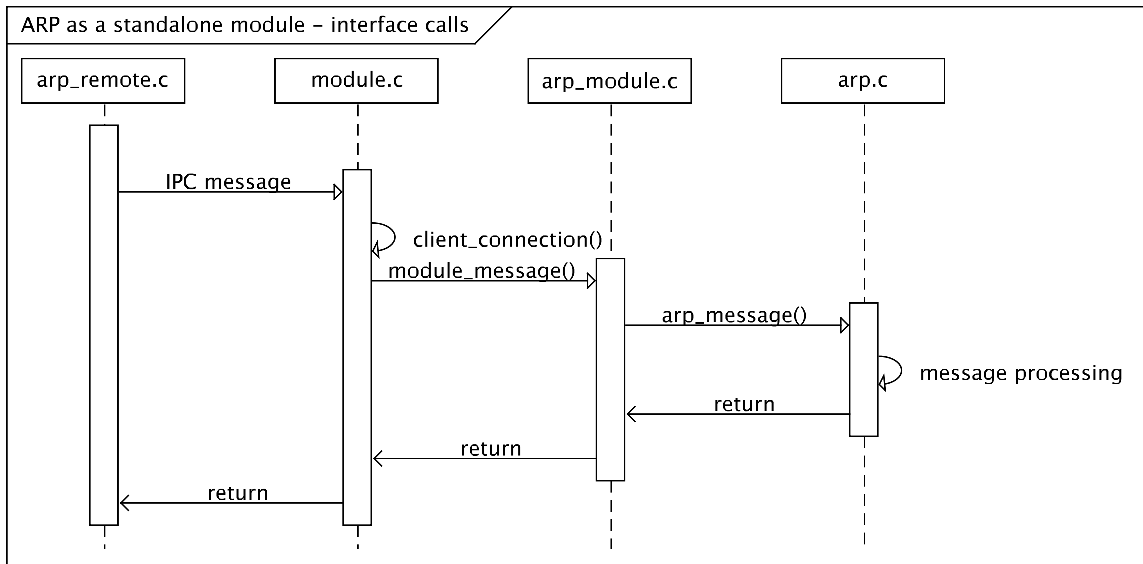
Networking stack module overview

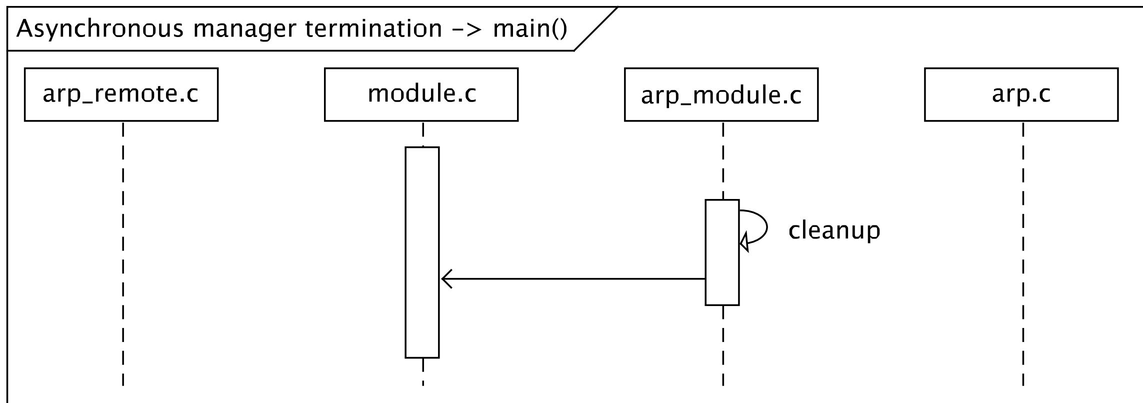


B.2 Module parts interaction

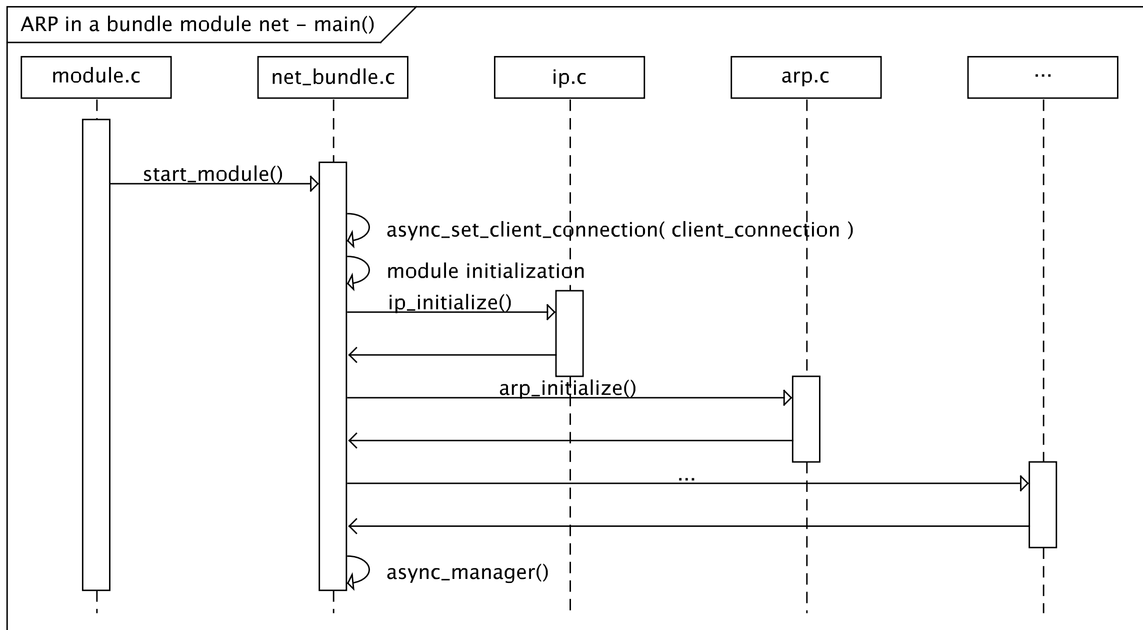
Standalone module

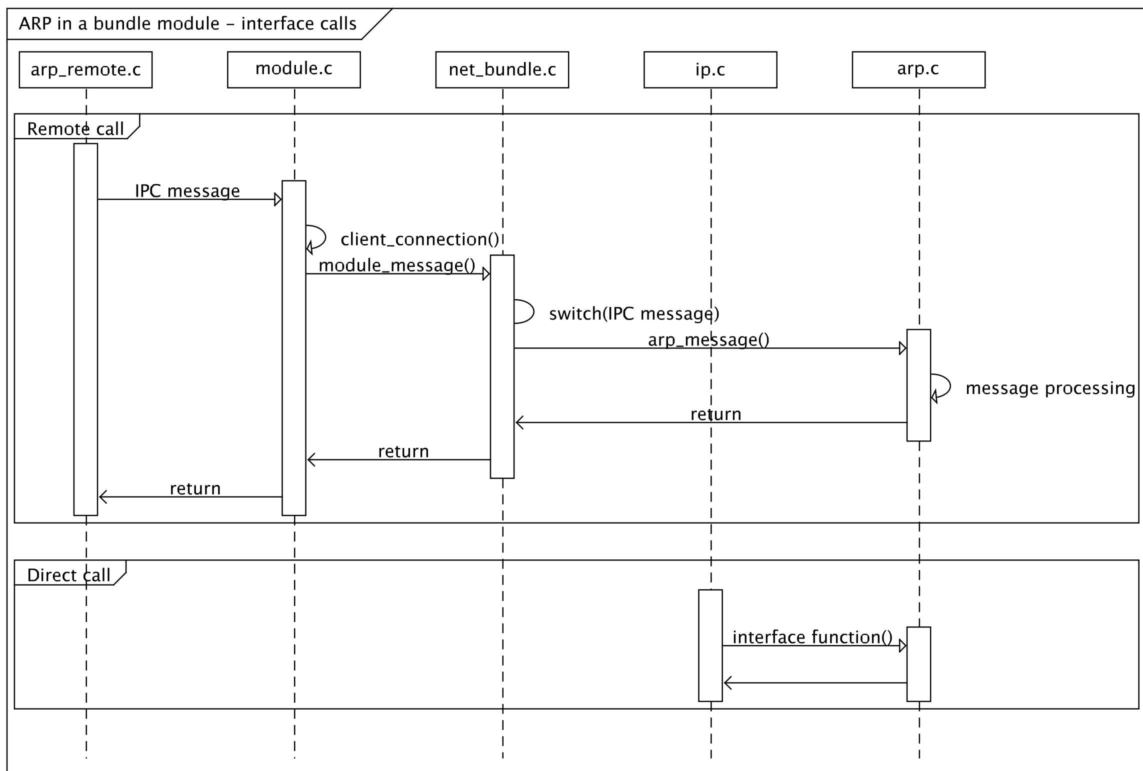






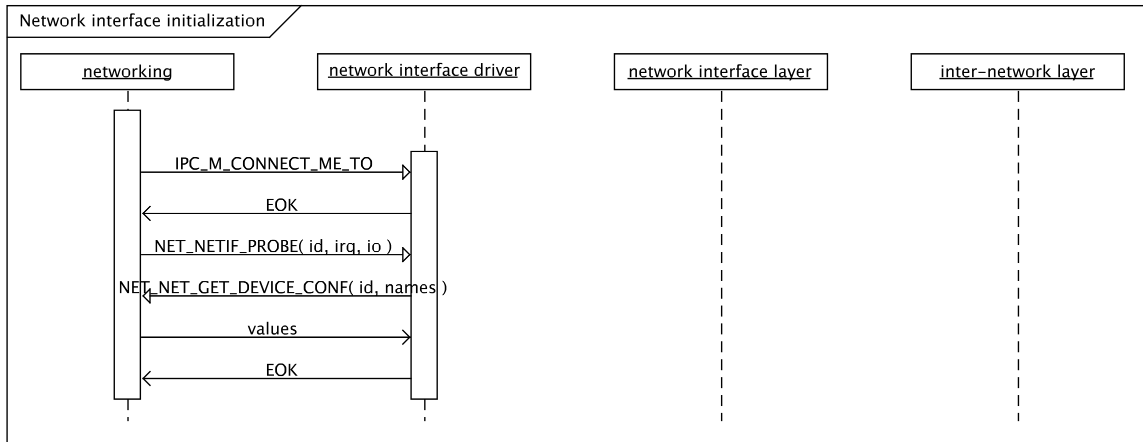
Bundle module



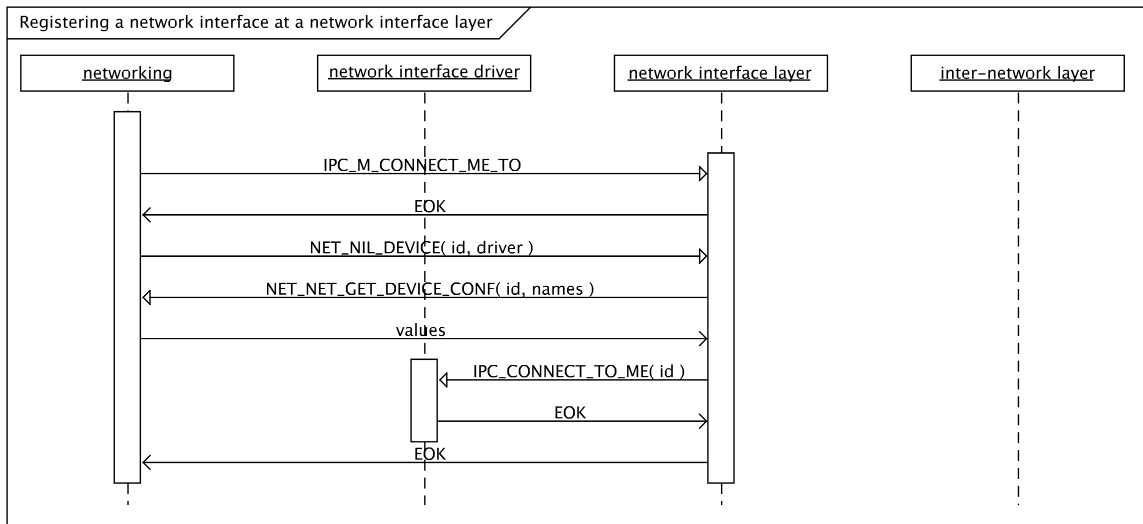


B.3 Network interface initialization

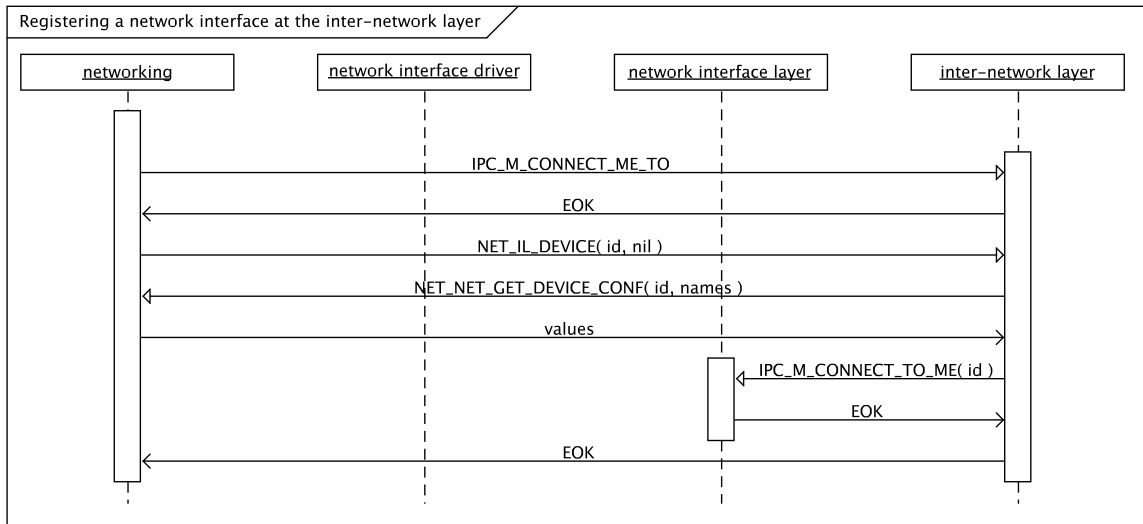
Network interface driver



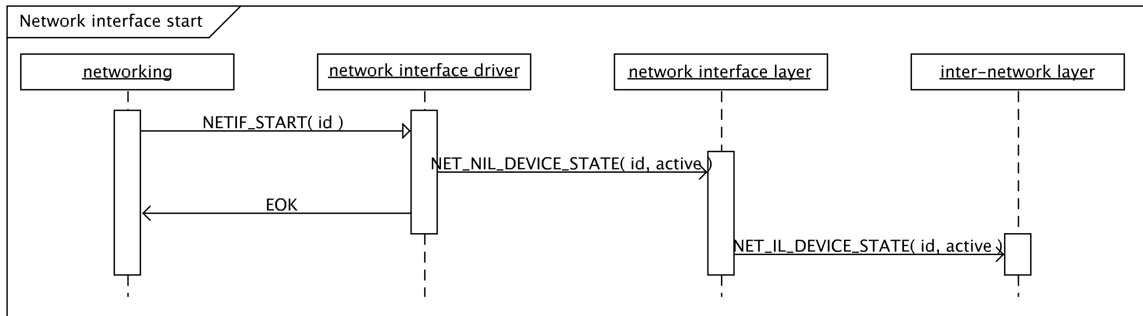
Network interface layer



Inter–network layer

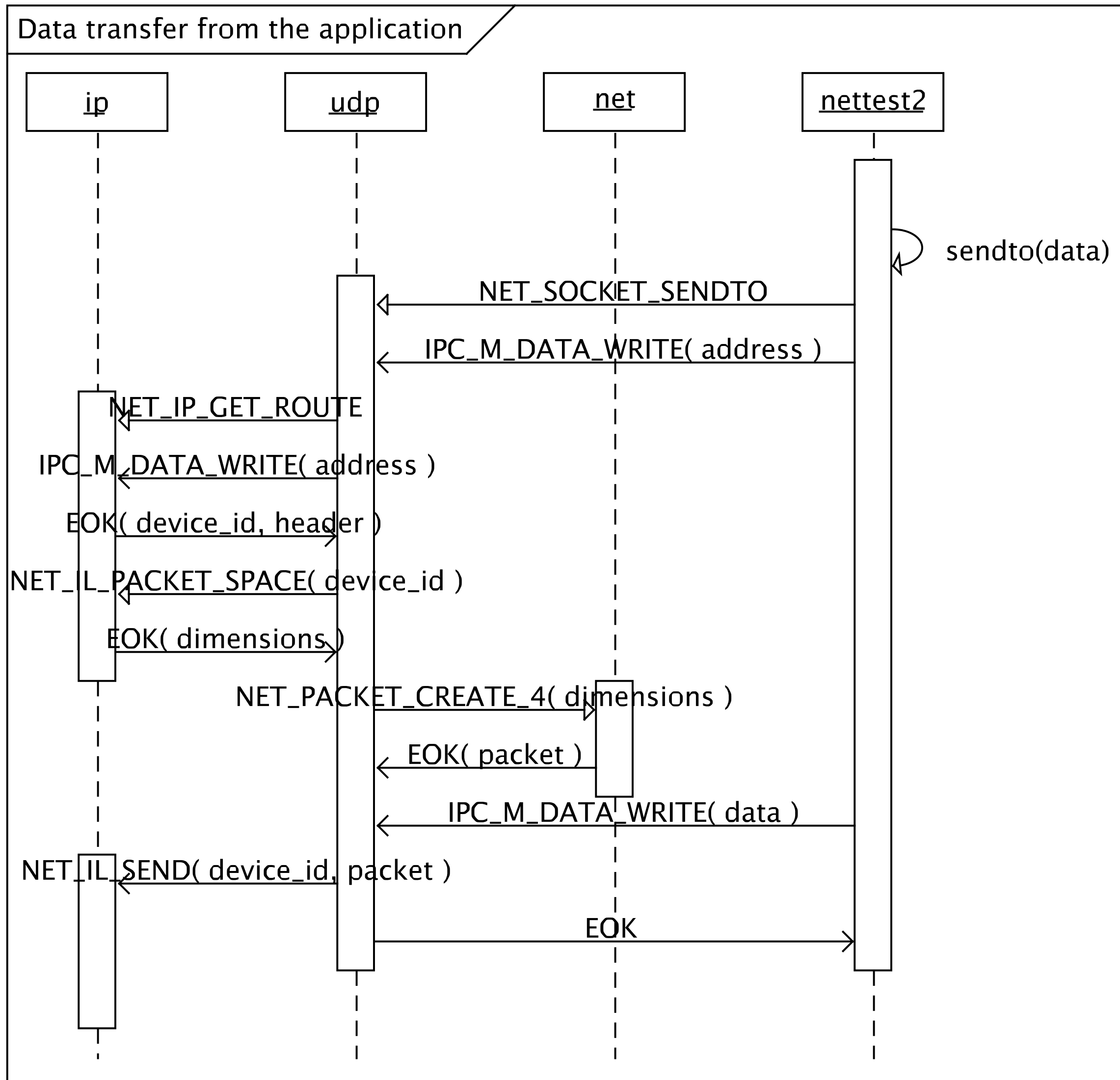


Network interface start

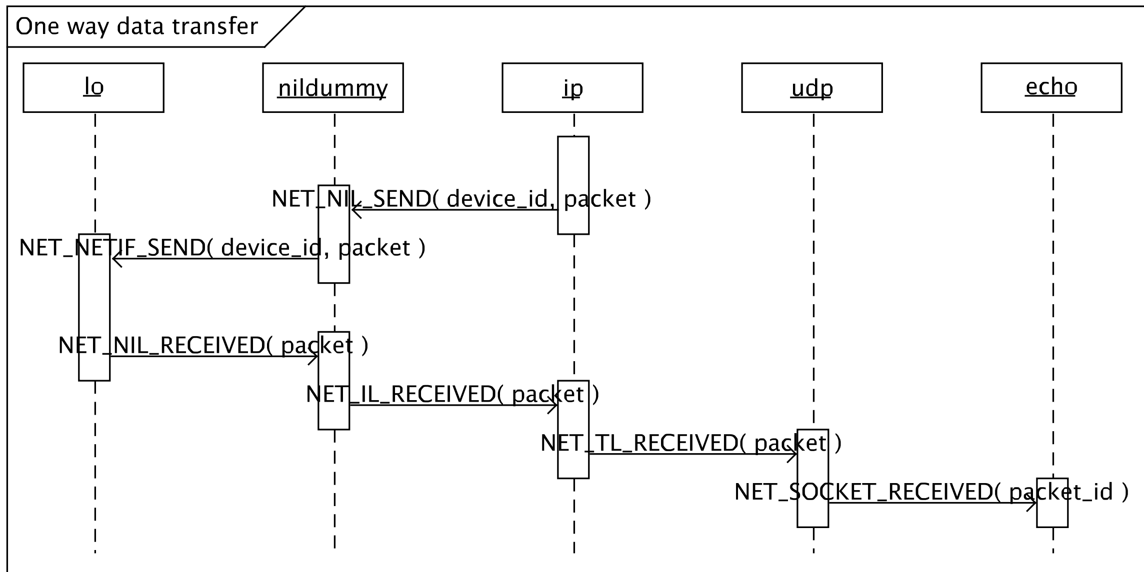


B.4 Data transfers

Data transfer from the application



One way data transfer



Data transfer to the application