UNIVERSITÄT ZU LÜBECK
INSTITUT FÜR MULTIMEDIALE
UND INTERAKTIVE SYSTEME

Titel der Arbeit

Masterarbeit/Bachelorarbeit

im Rahmen des Studiengangs
Medieninformatik
der Universität zu Lübeck

vorgelegt von:
Vorname Nachname

ausgegeben und betreut von:
Titel Name Erstgutachter

mit Unterstützung von:
Titel Name Betreuer

[Optional:] Die Arbeit ist im Rahmen einer Tätigkeit bei der Firma XYZ entstanden.

Lübeck, Tag. Monat. Jahr

Abstract

Der Abstract einer Abschlussarbeit sollte eine kurze Zusammenfassung enthalten, damit der Leser nach einigen Sätzen einen Eindruck davon bekommt, welches Thema bearbeitet wurde. Ein Abstract ist dabei kein “Teaser” sondern eher eine “Executive Summary”.
Dieses Dokument dient als Vorlage und gleichzeitig als kleine Anleitung, um eine Abschlussarbeit mit LaTeX zu erstellen. Um das Template für die eigene Abschlussarbeit zu verwenden, kann einfach der vorhandene Text gelöscht und eigener Text hinzugefügt werden. Das Dokument enthält keine ausführliche Erklärung für das Arbeiten mit LaTeX, da es hierzu eine Vielzahl von Tutorials im Internet gibt. Stattdessen enthält es einige Tipps und Richtlinien. Der Quellcode ist ausführlich dokumentiert, damit es einfach ist das Template für die eigene Arbeit anzupassen.

Contents

1 Introduction

1.1 Motivation

1.2 Related Work

1.3 Research Questions

2 Background

2.1 Positioning in the distributed Test Support System

2.2 TCP/IP Reference Model

Protocols are the basis for communication between instances in a network. They specify rules that must be followed by all communication partners [33]. Reference models arrange protocols hierarchically in layers. Each layer solves a specific part of the communication task and uses the services of the layer below while providing certain services to the layer above [36].

Figure 2.1 illustrates the relationship between service and protocol. A service refers to a set of operations that a layer provides to the layer above it, and it defines the interface between the two layers [33].

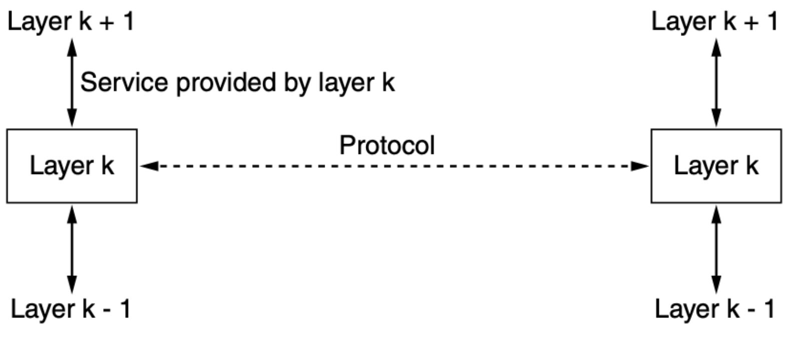


Figure 2.1: Relationship between service and protocol. Source: [33].

A protocol is a set of rules that define the format of messages exchanged within a layer [33]. These rules define the implementation of the service offered by the layer The transparency principle applies, meaning that the implementing protocol is transparent to the service user and can be changed as long as the service offered remains unchanged [36].

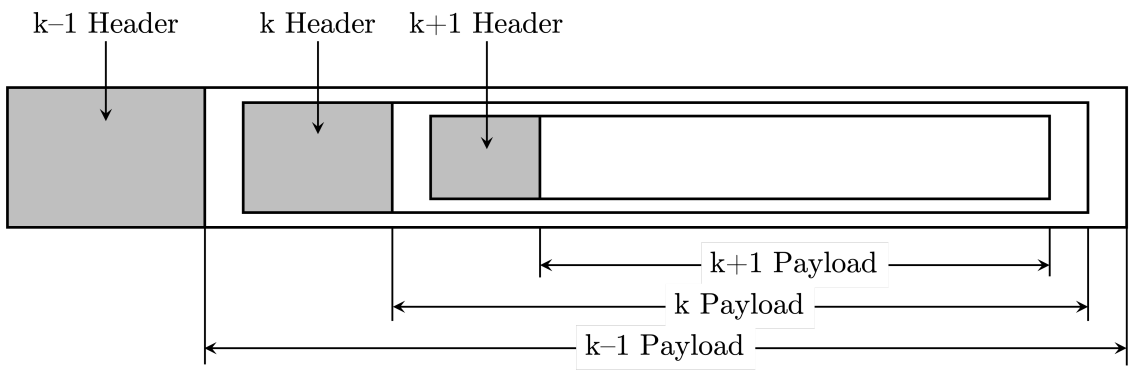


Figure 2.2: Encapsulation Principle. Adapted from: [33].

Protocols define the format of control information required by layer k to provide the service. This information is attached as a header or trailer to the data of layer k + 1, known as the payload, and is removed by the receiving instance. This principle is known as the 'Encapsulation Principle' and is illustrated in Figure 2.2 [33].

2.2.1 Introduction of the Reference Model

The following section presents a explanation of the TCP/IP reference model. Throughout this section, we will refer to the hybrid reference model proposed by Andrew S. Tanenbaum in [33]. Figure 2.3 shows this hybrid reference model. The physical layer is at the bottom, and the application layer is at the top. The tasks of each layer are briefly described here. For additional information, please refer to [33].

- The **Physical Layer** serves as the interface between a network node and the transmission medium, responsible for transmitting a bit stream. This involves line coding, which converts binary data into a signal. Additionally, the physical layer encompasses the transmission medium and the connection to this medium [33, 36].

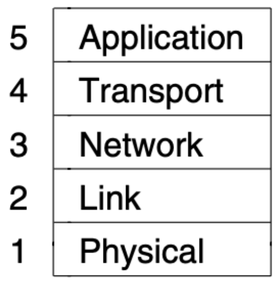


Figure 2.3: Hybrid TCP/IP Reference Model. Source: [33].

- The **Link Layer** facilitates reliable transmission of a sequence of bits (called a frame) between adjacent network nodes. This encompasses frame synchronization, which involves detecting frame boundaries in the bit stream, error protection, flow control, channel access control, and addressing [36].
- The **Network Layer** provides end-to-end communication between two network nodes. This includes addressing and routing [33, 36].
- The **Transport Layer** provides the transfer of a data stream of any length between two application processes. This involves collecting outgoing messages from all application processes and distributing incoming messages to them [36].
- The **Application Layer** serves as the interface to the application. It is responsible for implementing protocols for network use, such as file transfer or network management [36].

2.2.2 Protocols of the Reference Model

Figure 2.4 shows a selection of important protocols of the TCP/IP reference model including their assignment to the respective layer. The illustration also shows the dependency of the protocols on each other.

In this section, the characteristics of the protocols TCP, UDP, IP and Ethernet (IEEE 802.3), which are relevant for this work, are explained in detail according to [33]. Further information about the protocols of the TCP/IP reference model can be found in [33].

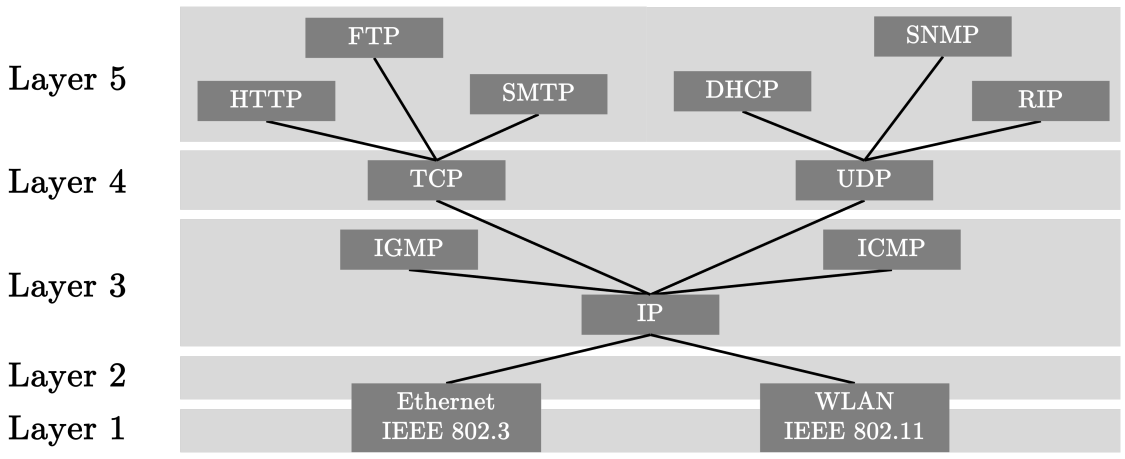


Figure 2.4: Selection of important protocols of the hybrid TCP/IP Reference Model. Adapted from: [36].

2.2.2.1 Ethernet (IEEE 802.3)

Ethernet, as defined by IEEE standard 802.3, specifies both hardware and software for wired data networks. This means that Ethernet includes both the physical layer and the link layer of the presented hybrid TCP/IP reference model.

2.2.2.1.1 Ethernet Physical Layer

The Ethernet physical layer consists of a number of standards that define different media types associated with different transmission rates and cable lengths.

Ethernet defines physical layer standards with transmission rates ranging from 10 Mbit/s to 1.6 Tbit/s, which is currently under development as the 802.3dj standard [9]. Both fiber and copper are used as transmission media. In the following, the 802.3an standard will be briefly discussed, since it is the one that will be used most in this thesis.

The 802.3an standard was published in 2006 and defines data transmission with a transmission rate of 10 Gbit/s over twisted-pair cables [6], also referred to as 10 GbE. Twisted-pair cables are copper cables in which pairs of copper wires are twisted together to reduce electromagnetic interference. Twisted-pair cables are divided into categories based on various characteristics, such as shielding or twist strength [1]. For 802.3an, a maximum cable length of 100 meters is specified in conjunction with Cat7 cables. 802.3an specifies the RJ45 connector as the plug connector.

According to 802.3an, the PAM16 line coding is used for Ethernet at 10 Gbps. It uses the principle of pulse amplitude modulation, which is described in detail in [7]. PAM16 allows the transmission of data by varying the amplitude of a signal in 16 different stages. Each stage represents four bits of information.

In addition to 802.3an, the Ethernet physical layer according to 802.3ae was also used in this work. This also defines the physical layer with a transmission rate of 10 Gbit/s. However, fiber optic cables are used in conjunction with transceiver modules called SPF+ SR [6].

2.2.2.1.2 Ethernet Link Layer

At the link layer, Ethernet defines frame formatting, addressing, error detection, and access control. This is also called the Medium Access Control (MAC) sublayer.

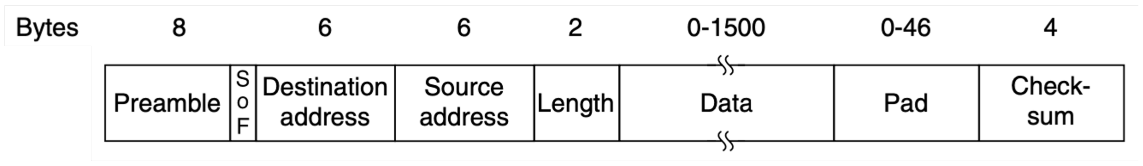


Figure 2.5: Structure of the Ethernet frame. Source: [33].

Figure 2.5 shows the IEE 802.3 frame format. The Ethernet header consists of the fields '*Destination address*', '*Source address*' and '*Length*' and therefore has a size of 14 bytes.

Each frame begins with a *preamble.* This has a length of 8 bytes and contains the bit sequence 10101010. An exception is the last byte, which contains the bit sequence 10101011 and is referred to as the *Start of Frame* (SoF). The preamble is used for synchronization between the sender and receiver. The last byte of the preamble marks the start of a frame [33].

This is followed by the *destination* and *source address.* This is the MAC address, which is uniquely assigned globally to a network interface [36]. This consists of a manufacturer code with a length of 3 bytes, followed by the serial number of the network interface, which also has a length of 3 bytes. The MAC address enables the Ethernet protocol to uniquely identify a station in the local network.

The *Length* field specifies the length of the next data field. In IEEE 802.3 Ethernet, this has a maximum length, called the Maximum Transfer Unit (MTU), of 1500 bytes. However, there are Ethernet implementations that use a larger MTU than specified in the original standard. These are known as jumbo frames [37].

In addition to a maximum length, the Ethernet standard also specifies a minimum length. An entire Ethernet frame must therefore have a minimum length of 64 bytes from the destination address to the checksum. To ensure that this can be achieved even with a small data field, padding information is added. The specification of the minimum length is related to the access control used.

The Ethernet frame ends with a 4-byte *checksum* that is used for the Cyclic Redundancy Check (CRC) based on polynomial divisions, as explained in [33]. This checksum serves to detect errors during transmission.

Ethernet originally used a shared transmission medium, allowing multiple communication participants to use it simultaneously. To control access, the MAC sublayer employs the CSMA/CD algorithm, ensuring that only one device transmits data at a time. Each device listens to the medium (carrier sense) before sending data to determine whether it is free. It also performs collision detection to determine whether two devices have started sending at the same time. In such a case, the devices stop the transmission and retry it after a random waiting time to avoid the collision [33].

The 802.3an specification for 10 Gigabit Ethernet is exclusively for point-to-point full-duplex connections, which eliminates the need for access control such as CSMA/CD. As a result, it is no longer included in the specification [24].

In order to connect multiple network devices with point-to-point connections, Ethernet switches are used. They have multiple ports and forward packets based on the MAC address. Ethernet switches operate on layer 2 of the reference model.

To summarise, with Ethernet there is no guarantee that data will be transmitted reliably and without loss. Although Ethernet uses CRC for error detection, faulty frames are generally discarded. Additionally, Ethernet does not provide flow control or overload detection, which must be performed by a higher layer.

2.2.2.2 IP

The Internet Protocol (IP) is a central protocol in the TCP/IP reference model. Its tasks include connecting different networks, addressing network participants, and fragmenting packets [36]. IP is a connectionless protocol that operates on the 'best effort' principle, meaning it does not guarantee delivery.

There are two versions of the Internet Protocol: IPv4 and IPv6. As this work uses the IPv4 protocol, it is presented in more detail below.

2.2.2.2.1 IP Header

The IPv4 datagram is divided into a header and a payload. The header typically spans 20 bytes, but may also include an optional variable-length section. The header is shown in Figure 2.6.

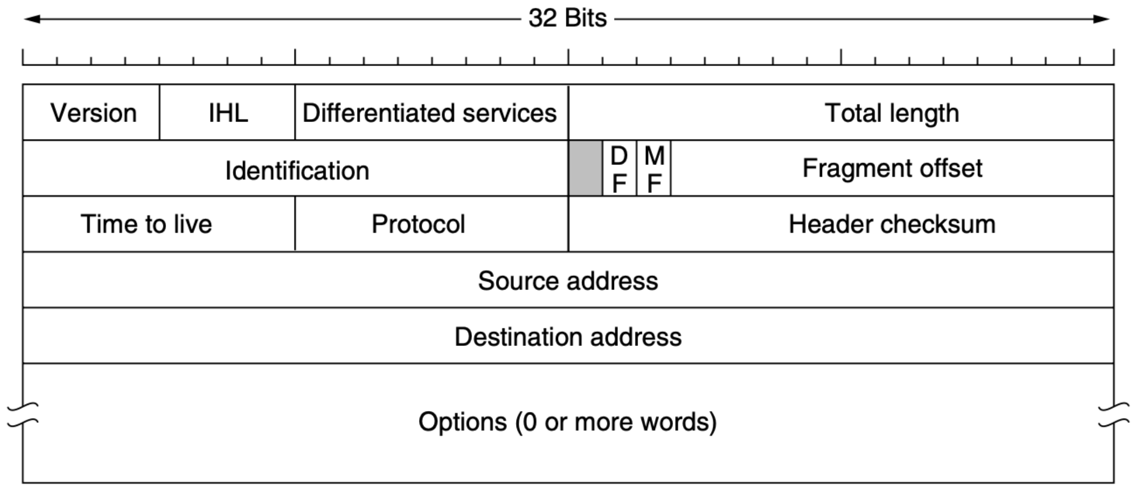


Figure 2.6: Structure of the IP Header. Source: [33].

The first field in the header is the 4-bit *Version* field. This indicates the IP version used. For IPv4, the value is always 4.

The *IHL* (Internet Header Length) field specifies the number of 32-bit words in the header. This is necessary because the header can contain options and therefore has a variable length. The minimum value of the field is 5 if there are no options.

The *Differenciated Services* field specifies the service class of a packet, allowing for prioritisation of certain data traffic using Quality of Service (QoS). For a more detailed description of Quality of Service, please refer to section 2.5.2.

The *Total Length* field indicates the total length of the datagram, including the header. Due to the field size of 16 bits, the maximum length is 65535 bytes. However, a packet's length is also limited by the Layer 2 MTU [36], resulting in datagrams being split into multiple packets, known as fragmentation.

The *Identification* field is assigned a number by the sender, which is shared by all fragments of a datagram.

A flag field with a length of 3 bits follows, with the first bit being unused. The second section includes the 'Don't Fragment' (*DF*) flag, which indicates that intermediate stations should not fragment this packet. The third section contains the 'More Fragments' (*MF*) flag, which indicates whether additional fragments follow. This flag is set for all fragments except the last one of a datagram.

The *Fragment Offset* field specifies the position of a fragment in the entire datagram.

The *Time to live* (TTL) field specifies the maximum lifetime of a packet. The TTL value is measured in seconds and can be set to a maximum of 255 seconds. This is done to prevent packets from endlessly circulating in the network.

The *Protocol* field identifies the Layer 4 protocol used for the service. This allows the network layer to forward the packet to the corresponding protocol of the transport layer. The numbering of the protocols is standardized throughout the Internet.

The *Header checksum* field contains the checksum of the fields in the IP header. The IP datagram's user data is not verified for efficiency reasons [12]. The checksum is calculated by taking the 1's complement of the sum of all 16-bit half-words in the header. It is assumed that the checksum is zero at the start of the calculation for the purpose of this algorithm.

The two 32-bit fields *Source Address* and *Destination Address* contain the Internet Protocol address, called the IP address. Section 2.2.2.2.2 provides further details on this topic.

The *Options* field can be used to add additional information to the IP protocol. For example, there are options to mark the route of a packet.

2.2.2.2.2 IP addresses and routing

This section provides a brief description of the structure and important properties of IP addresses. The network examined in this thesis is an isolated local network that is not connected to other networks. As a result, the network layer does not perform any routing based on IP addresses. For further information on routing, please refer to [33].

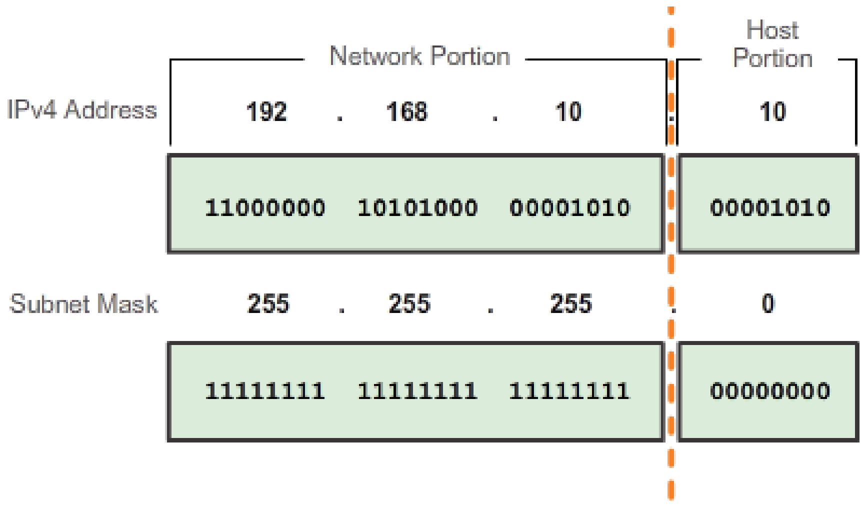


Figure 2.7: Structure of the IP address and subnet mask. Source: [27].

Every participant on the Internet has a unique address, known as an IP address. This has a total length of 32 bits and a hierarchical structure that divides the IP address into a network portion and a host portion. The division between the two parts is variable and is defined by a so-called subnet mask, which is illustrated in Figure 2.7. The bits of the network portion of the IP address are marked with ones.

- The **network portion** identifies a specific network, such as a local Ethernet network, and is the same for all participants in this network.
- The **host portion** identifies a specific device within this network.

Routing, which is another important task of the network layer, is based on IP addresses. The packet can be directed to its destination using the network portion of the IP address. The path to the destination is determined by specific routing algorithms. As mentioned earlier, the thesis only considers an isolated local network, so further discussion on routing will be omitted.

2.2.2.2.3 Address Resolution Protocol

The Address Resolution Protocol, abbreviated to ARP, is an auxiliary protocol of the network layer. Its task is to map the IP addresses to a MAC address and vice versa, as the sending and receiving of data in the underlying link layer is based on these MAC addresses [36].

2.2.2.2.4 Fragmentation and Defragmentation

As explained in 2.2.2.1.2, the link layer defines a maximum data size known as MTU. Since IPv4 datagrams have a maximum size of 65535 bytes, they must be divided into smaller packets, or fragments, each with its own IP header.

The IP header (refer to Figure 2.6) contains information necessary for the target system to assemble fragmented packets, a process known as defragmentation. This includes the ID that assigns all fragmented packets to a datagram, as well as the fragment offset that specifies their position within the datagram. The 'More Fragment' flag indicates whether additional fragments will follow.

Fragmentation has the advantage of allowing IPv4 datagrams larger than the MTU to be sent, but the disadvantage is that the loss of a single fragment results in the loss of the entire datagram. Additionally, fragmentation can cause packet reordering [17].

2.2.2.3 TCP and UDP

TPC and UDP are transport layer protocols. As a service, they provide the transmission of a data stream of any length between two application processes. The services of the network layer are used for this purpose.

2.2.2.3.1 TCP

TCP provides **reliable** transmission of a byte stream in a **connection-oriented** manner. A virtual connection is established between the two instances before transmission, which is terminated after transmission.

TCP also implements flow control to ensure reliable data transfer between sender and receiver without losses and to prevent overloading at the receiver. TCP provides congestion control to prevent network overload and ensures reliable transmission using Positive Acknowledgement with Re-Transmission (PAR) algorithm [13].

TCP is known for its secure data transmission. However, it requires a significant amount of control information to implement its functions. The Transmission Control Protocol (TCP) header is 20 bytes in size. In addition to an application identifier (port number), it contains flow control and congestion control information. This overhead can negatively impact transmission speed. Additionally, the data loss from the underlying layers combined with the flow control used by TCP leads to delays and reduced throughput, which can have a significant impact on the performance of the application.

2.2.2.3.2 UDP

In contrast to TCP, UDP is an **unreliable** and **connectionless** protocol. The protocol sends packets, called datagrams or segments, individually. UDP lacks mechanisms for detecting the loss of individual datagrams, and the correct sequence of these is not guaranteed.

Figure 2.8 displays the UDP header, which has a size of 8 bytes. It is considerably smaller than the TPC header, which has a size of 20 bytes.

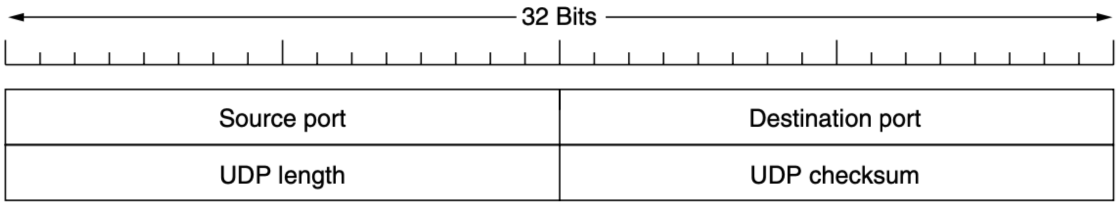


Figure 2.8: Structure of the UDP Header. Source: [33].

The header includes the fields *Source port* and *Destination port* to identify the endpoints in the respective instance. When a packet arrives, the payload is passed to the application using the appropriate port number via the UDP protocol.

The *UDP length* field indicates the length of the segment, including the header. The maximum length of data that can be transmitted via UDP is limited to 65,515 bytes due to the underlying Internet Protocol.

The last field of the header is a 16-bit *UDP checksum*. This checksum is formed via the so-called IP pseudoheader, which contains the source and destination IP address, the protocol number from the IP header, and the *UDP length* field of the UDP header.

Compared to TCP, UDP can achieve higher data transmission speeds due to its lower protocol overhead, ase the UDP header is only 8 bytes in size. Furthermore, UDP does not require an acknowledgement of the transport or other mechanisms used by TCP to provide a reliable connection. This makes it very efficient and reduces processing overhead.

2.3 Linux Kernel

The Linux kernel is an operating system kernel that is available under a free software license and has been under development since 1991 [35]. The Linux kernel is the main component of a Linux operating system and is used by a large number of operating systems, called distributions. Popular examples of such distributions are Ubuntu or Linux Mint, which are used in this thesis.

This chapter will take a closer look at the Linux kernel. However, due to the scope of the Linux kernel, readers are referred to [3], [19] and [23], which provide a detailed and comprehensible insight into the Linux kernel. Additionally, a basic knowledge of operating systems is required, which can be obtained from [10].

An operating system kernel serves as the interface between the hardware and the processes of a computer system [28]. It manages hardware resources, schedules processes, and facilitates communication between application software and hardware [25].

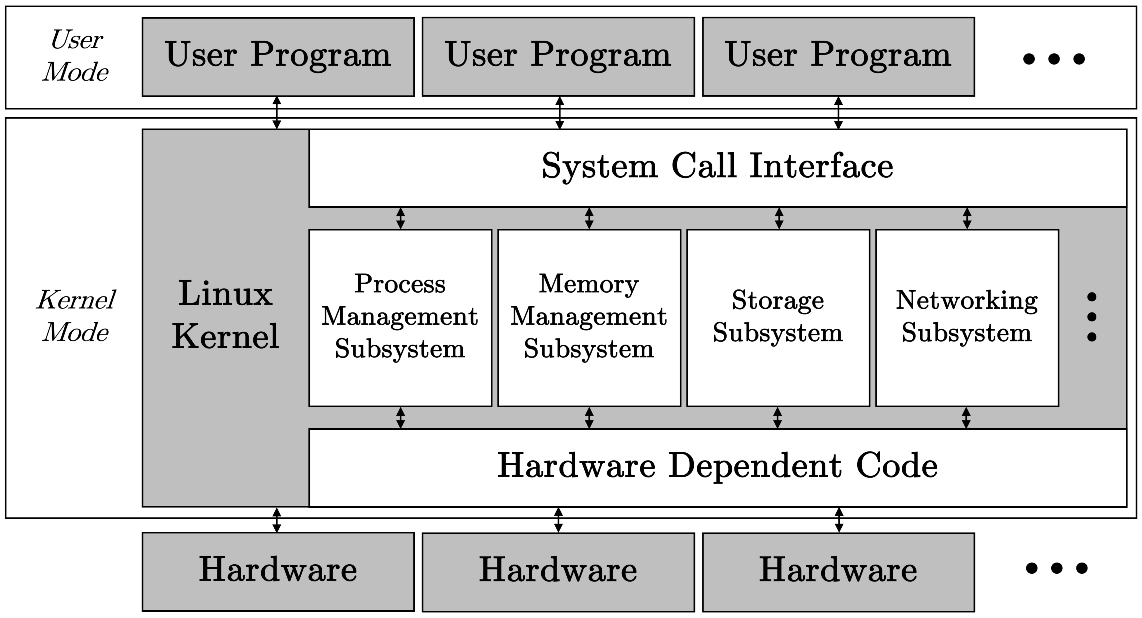


Figure 2.9: Simplified representation of the Linux Kernel with selected Subsystems.

Figure 2.9 presents a condensed overview of the architecture of a Linux operating system. The illustration highlights some selected features of the kernel.

Linux consists of a monolithic kernel. This means that the entire kernel is implemented as a single program, and all kernel services run in a single address space. Communication within the kernel is achieved through function calls [3].

This is in contrast to the microkernel, which divides functionalities into separate modules and uses message passing for communication between them. Although the Linux kernel is based on a monolithic approach, it adopts some aspects of a microkernel, such as a modular architecture with different subsystems or the ability to load modules dynamically. However, communication within the kernel occurs through function calls, which provides better performance compared to message passing [23].

In the following, the characteristics of the Linux kernel and its environment shown in Figure 2.9 are described.

2.3.1 User Mode and Kernel Mode

The Linux architecture distinguishes between two basic execution environments: user mode and kernel mode. Application processes run in user mode with restricted rights, while the Linux kernel, which is the main part of the operating system, runs in kernel mode [3].

This separation requires corresponding support in the processor. This system monitors aspects such as memory access, branches, or the executed instruction set in user mode and intervenes in the event of unauthorized access, for example, by stopping the process [25]. The transition between the different execution environments occurs as part of a system call.

2.3.2 System Call Interface

Processes that request a service from the Linux kernel use system calls. These calls are made through a software interrupt (trap), which causes the CPU to switch to kernel mode and call the so-called system call handler. In the handler, the requested service is identified using an ID transmitted by the user process, and the corresponding instructions are invoked [3]. A process or application executes a system call in kernel space. This is also referred to as the kernel running in the context of the process.

In the monolithic kernel, individual instructions call other instructions of the kernel. This sequence of instructions, executed during a system call, is referred to as the *kernel control path* [34].

The Linux kernel is a *reentrant kernel.* Several processes can be executed simultaneously in kernel mode, which also means that the process can be interrupted while instructions are being executed in kernel mode. Functions in a reentrant kernel should therefore only change local variables and not affect global data structures. However, there are also non-reentrant functions in the kernel, for which corresponding locking mechanisms are used [3].

It should be noted here that system calls are not the only way to execute instructions in the kernel. According to [3], there exist other ways besides system calls:

- A exception is reported by the CPU, which are handled by the kernel for the originating process. An example of this is the execution of an invalid instruction.
- A peripheral device sends an interrupt signal to the CPU, which is processed by a function called the interrupt handler. As peripheral devices work asynchronously to the CPU, interrupts occur at unpredictable times.
- A kernel thread is executed. These run in kernel mode and are mainly used to perform certain tasks periodically.

2.3.3 Hardware and Hardware Dependent Code

As already mentioned, the kernel is the interface between the hardware and the processes of a system. Many operations in the kernel are related to the access of physical hardware.

The Linux kernel distinguishes between three different types of hardware devices [8]:

- **Block Devices** – devices with block-oriented addressable data storage (e.g. hard drives)
- **Character Devices** – devices that handle data as a stream of characters or bytes (e.g. keyboards)
- **Network Devices** – devices that provide access to a network

The abstraction layer between the physical hardware and the Linux kernel are device drivers. Their primary function is to initialize the device and register its capabilities with the kernel. Additionally, drivers enable the kernel to access, control, and communicate with the device. Each driver is specific to a device and implements certain predefined interface functions to the Linux kernel, depending on the type of the device. The device drivers are available as modules that can be loaded dynamically at runtime [5].

One way for the physical hardware to interact with the kernel via the device drivers is through interrupts, which is referred to as interrupt-driven I/O. The hardware uses *interrupt requests* to inform about certain events, and the driver implements the associated *interrupt handler* to process the request [5].

Direct Memory Access (DMA) is another way of interaction between the hardware and a system, which is mainly used by block devices or network devices [9]. DMA is a mechanism that allows hardware devices to transfer data directly to or from system memory, bypassing the CPU. This method enhances data throughput and system performance, as it reduces CPU overhead during high-volume data transfers [10].

Additional information regarding the interface between the Linux kernel and hardware, as well as device drivers, can be found in [5].

2.3.4 Kernel Subsystems

As previously stated, the Linux kernel is a monolithic kernel that is subdivided into various subsystems. A subsystem is a group of functions that work together to perform a specific task [22]. Figure 2.9 displays the most significant subsystems, which are further explained below based on [19] and [5].

2.3.4.1 Process Management Subsystem

The process management subsystem is responsible for the administration of processes. This task can be divided into three main parts:

- Creation and termination of processes and their related resources
- Communication between different processes (e.g. with *Signals* or *Pipes*)
- Scheduling

2.3.4.2 Memory Management Subsystem

The primary function of the memory management subsystem is *Virtual Memory Management*. This enables more efficient use of RAM. Each process is assigned a virtual address space, and parts of it that are not currently required can be swapped to disk. This is based on the locality principle of programs. Additionally, *Virtual Memory Management* enables isolation between processes.

The memory management subsystem in the Linux kernel provides memory for other kernel modules, for example through malloc/free operations.

2.3.4.3 Storage Subsystem

The storage subsystem is responsible for creating and managing the file system on the physical media, such as the disk.

2.3.4.4 Networking Subsystem

The networking subsystem handles the sending and receiving of packets in networks and their distribution to applications in user space. Additionally, it implements network protocols such as those used by the TCP/IP protocol stack presented in 2.4.1.2.1.

A detailed description of specific parts of the networking subsystem can be found in chapter 2.4.

2.4 UDP communication with a Linux Operating System

The purpose of this chapter is to explain UDP communication using a Linux operating system. The fundamental processes are described, with an emphasis on the interaction among the different components.

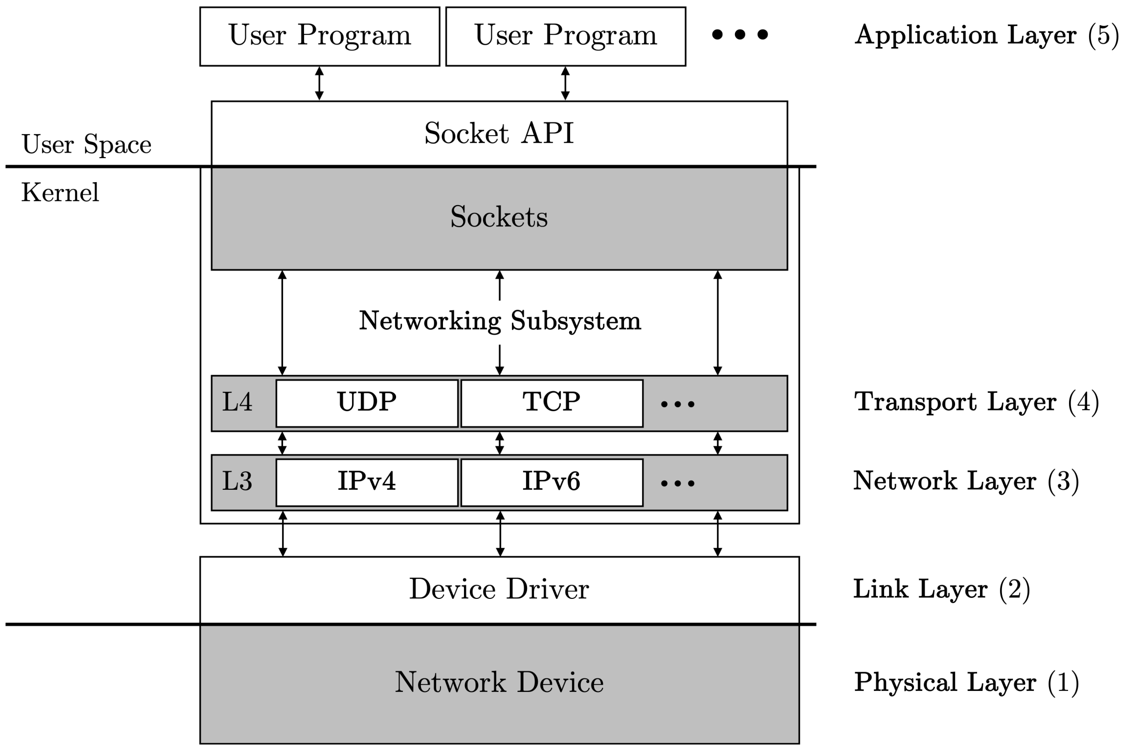


Figure 2.10: Simplified Representation of the Linux Network Stack with associated Layers of the hybrid TCP/IP Reference Model. Adapted from: [2].

Figure 2.10 presents a simplified schematic of the components of the Linux network stack and how they relate to the layers of the hybrid TCP/IP reference model presented in chapter 2.2. This section provides a simplified representation, based on [2], that illustrates the relationship between the components of the network stack.

First, the components of the network stack will be presented, with a focus on the protocols represented in the TCP/IP reference model. Then, the interaction between the components will be explained by following the path of a packet through the network stack during transmission and reception.

2.4.1 Components in the Linux Network Stack

2.4.1.1 Sockets and the Socket API

Sockets are objects in the operating system that allow data to be exchanged between two applications, usually on a client-server basis. Data can also be exchanged across computer boundaries. Sockets are part of the networking subsystem in the Linux kernel. The socket API represents the associated programming interface [8][18].

Sockets serve as the interface between the application layer and the transport layer in the Linux kernel. Sockets can be defined as the endpoints of a communication channel between two applications. They do not form a separate layer, but allow the application to access the services of the underlying layer, usually the transport layer. The operating system manages all sockets and their associated information [16].

2.4.1.1.1 Characteristics of Sockets

A socket is a generic interface that supports various protocols and protocol families, also known as communication domains. This section focuses on sockets for the TCP/IP protocol family, also known as Internet sockets [19].

2.4.1.1.1.1 Socket Descriptor

In line with the Linux philosophy of '*everything is a file*', sockets in a system are also represented by an integer, called a socket descriptor in this context. This descriptor can be obtained through a specific call to the operating system and can be used to perform operations such as `write()` or `read()`, similar to handling files. Additionally, there are specialized methods like `send()` or `receive()` that provide further options.

2.4.1.1.1.2 Socket Types

There are different types of sockets that vary in their properties. The two most common types are stream sockets and datagram sockets [19].

- **Stream sockets** operate in a connection-oriented manner between a client and server application. A connection must be established between the partners before data can be transferred. The TCP protocol is used for this type for Internet sockets.

- **Datagram sockets** enable the exchange of individual messages. The sockets operate without a connection. The User Datagram Protocol (UDP) is utilized as the transport layer protocol, resulting in the provision of unreliable transmission.

Other socket types, such as raw sockets or packet sockets, also exist.

2.4.1.1.1.3 Socket Address

A socket can be identified externally using the socket address. In the context of Internet sockets, this address consists of the IP address and a port number and uniquely identifies the socket in the network [16].

2.4.1.1.2 Operation of Sockets

The following section presents important concepts and aspects of working with sockets. The focus is limited to connectionless datagram sockets, as this is the type of socket used in this thesis. For a detailed description of datagram sockets and stream sockets, please refer to [19], which serves as the basis for this section.

Figure 2.11 displays the system calls commonly used with a datagram socket for a client-server application. These calls are briefly described below:

- The `socket()` call requests the corresponding socket from the operating system, specifying the protocol family and socket type. The return value is the socket descriptor.

- The `bind()` call is used to bind the socket to a server address. For Internet sockets, the address consists of the IP address and the port of the server application. This enables the application to receive datagrams sent to this address.

- The client calls `sendto()` with both the data to be sent and the address of the

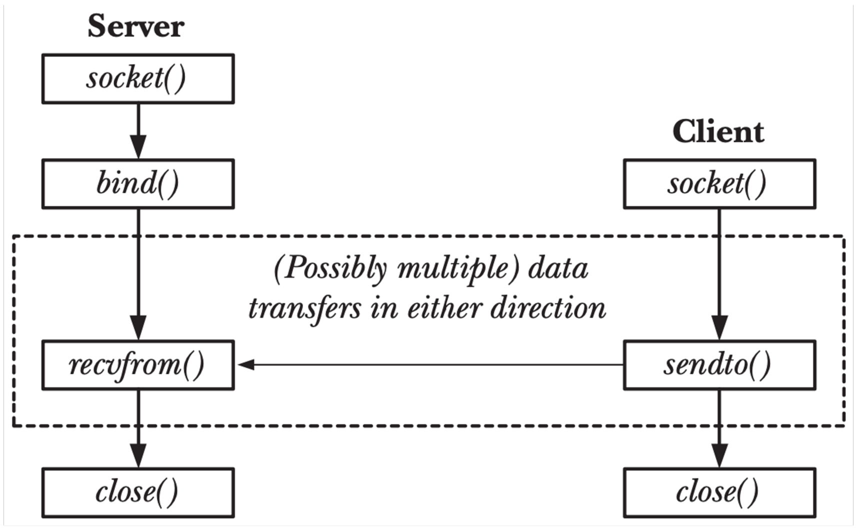


Figure 2.11: Overview of System Calls used with Datagram Sockets. Source: [19].

socket to which the datagram is to be sent. This call will send the data.

- To receive a datagram, `recvfrom()` is called. The argument can be used to specify the address of the sender's socket from which the data is to be received. If no restrictions should be defined for the sender's address, `recv()` can also be used.
  Both calls save exactly one received datagram in a buffer, a pointer to which is also passed as an argument to the function. If no data has been received when `recv()` or `recvfrom()` is called, the call is blocked.
  If multiple datagrams are received, they are stored in the receive buffer of the corresponding socket. However, when one of these functions is called, only one message is passed to the application via the socket.
- If the socket is no longer needed, it can be closed using `close()`.

2.4.1.1.3 Raw Sockets and Packet Sockets

Raw sockets and packet sockets are additional types of Internet sockets. These are an addition to the stream and datagram sockets already mentioned and allow access to lower layers of the network stack instead of hiding them from the user [11].

Figure 2.12 displays the access possibilities with different socket types. Raw sockets

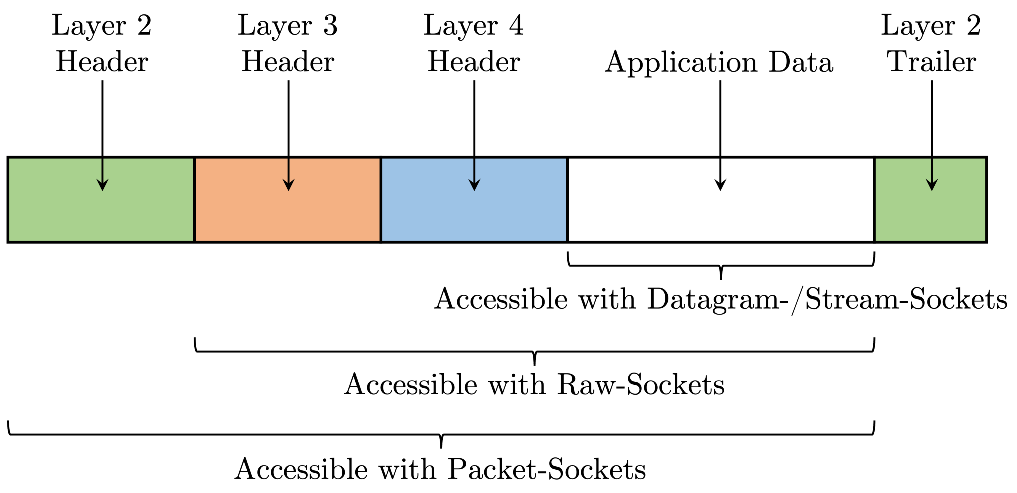

Layer 2 Header
Layer 3 Header
Layer 4 Header
Application Data
Layer 2 Trailer
Accessible with Datagram-/Stream-Sockets
Accessible with Raw-Sockets
Accessible with Packet-Sockets

Figure 2.12: Overview of Network Layers and Access Possibilities with different Socket Types. Adapted from: [11].

provide access to the transport layer (4) and network layer (3) of the network stack [21], including TCP or UDP as well as IP in the case of the TCP/IP reference model. Packet sockets can also be utilized to access the link layer (2), enabling access to almost the entire Ethernet frame, except for the preamble and trailer [20].

The concept underlying raw and packet sockets involves the implementation of separate protocol layers in the application. Depending on the chosen socket type, the application can implement layers 2 to 4 [21]. Furthermore, packet sockets can also be used to capture the entire communication of a system, as used by Wireshark, for example.

Raw or packet sockets eliminate the overhead of the respective protocol layer in the Linux kernel, which can potentially accelerate processing. They also increase flexibility, as certain fields in the header can be easily modified.

A disadvantage, however, is that in order to maintain compatibility with other TCP/IP implementations, the corresponding protocols must be fully and correctly implemented in the application. Additionally, using raw or packet sockets requires that the application be executed with root privileges.

The technical report 'Introduction to RAW sockets' [11] provides a comprehensive overview of raw and packet sockets and their application. This report was also used for programming in this thesis.

2.4.1.2 Layers 3 and 4 in the Networking Subsystem

The networking subsystem of the Linux kernel includes not only sockets but also layers 3 and 4, namely the transport and network layers. These layers implement the protocols described in , while sockets provide an interface between the application and the network stack.

2.4.1.2.1 Protocol Handler

The corresponding protocols are implemented in layers 3 and 4, including implementations for protocols from the TCP/IP reference model and other protocols in their respective layers. It is also possible to develop handlers for your own protocols [2].

An important task in the network subsystem is to execute the correct protocol handler for the corresponding layer. For outgoing packets, this is determined by the socket. For instance, an Internet socket that uses the datagram type employs the UDP protocol at layer 4 and the IPv4 protocol at layer 3. The protocol handler to be executed for incoming packets is determined from the header of the underlying layer. Both the Ethernet header and the IP header contain a corresponding field for the service-using protocol [2].

The protocol handlers of the respective layer implement the standardized behavior for this protocol. These are described in the chapter . Implementation details will not be discussed further at this point. For more information, please refer to [31].

2.4.1.2.2 Data Structures in the Networking Subsystem of the Linux Kernel

This chapter presents two significant data structures of the networking subsystem in the Linux kernel, as described in [2].

2.4.1.2.2.1 Socket Buffer Structure

The socket buffer structure, also known as `sk_buff`, is the most important data structure in the network stack. It represents a packet that has been received or is to be sent and is used by layers 2, 3, and 4. This structure eliminates the need to copy packet data between layers.

The structure contains control information associated with a network packet, but not the actual data itself. Included in this structure are:

- Information on the organization of the socket buffers by the kernel
- Pointers to the data and to the headers of layers 2, 3 and 4
- Length of the data and the headers
- Data on the internal coordination of the packet
- Information on the associated network device (see 2.4.1.2.2.2)

The mentioned pointers to the data point to a data field associated with the socket buffer. This field contains the packet data and associated headers and is created when a socket buffer is allocated. The socket buffer has pointers to different locations in this data field, depending on the layer currently using the socket buffer.

Additionally, there are management functions related to the socket buffers. These functions can be utilized by individual network layers to add or remove their headers to the packet during processing. There are also functions to modify the size of the data field.

2.4.1.2.2.2 Network Device Structure

The network device structure, also known as `net_device`, contains information about a specific network interface. This structure is present in the kernel for every network interface of the system.

Some important fields of this structure are (according to [31]):

- Identifier of the interface

- MTU of the network interface
- MAC address of the interface
- Configurations and flags of the interface
- Pointer to the transmit method of the interface

2.4.1.3 Network Device and Device Driver

The network device, also known as the network interface, along with its associated device driver, is the lowest component in the Linux network stack. The device driver performs the tasks of layer 2 of the TCP/IP reference model, while the network interface physically transmits the data, working on layer 1 of the reference model [31].

The main tasks of the device driver are to receive packets addressed to the system and forward them to layer 3 of the network stack, and to send packets generated by the system.

The driver interacts with the network interface, which transmits the data according to the respective transmission standard [31]. To exchange data with the interface, the driver creates two ring buffers: the TX_Ring and the RX_Ring, which are used for sending and receiving data. These buffers are located in the system memory and contain a fixed number of descriptors pointing to buffers where packets can be stored. They are empty during initialization and accessed by the interface via DMA [4, 15].

The implementation of the driver depends on the hardware and is therefore not standardized. However, the Linux kernel defines how the driver interacts with the networking subsystem.

2.4.2 Path of a Network Packet

This section explains how a packet travels through the Linux network stack by examining the interactions of the components described above. Specifically, this section will focus on the reception and transmission of a UDP packet.

The following overview provides a general understanding of the process and will serve as a foundation. For a more detailed explanation of the packet's path, please refer to [31] and [2].

2.4.2.1 Receiving a Packet

When a frame is received by the network card, it first checks for errors using the Ethernet frame checksum and then verifies if it is intended for the network interface by using the MAC address. The frame is then written to a free buffer in the RX_Ring via DMA, which was created by the device driver during initialization. If no buffers are available, the frame is dropped [4, 2, 15].

Interrupts are utilized to notify the system about a packet. Different strategies can be employed for this purpose. In the simplest case, a hardware interrupt is triggered for each received frame [2].

To minimize processing in the interrupt context, softirqs are utilized [4]. Softirqs are non-urgent interruptible functions in the Linux kernel that are designed to handle tasks that do not need to be done in the interrupt context. The handlers for the softirqs are executed by ksoftirq kernel threads, with one thread for each CPU core on the system [3].

The network driver's hardware interrupt handler schedules a softirq to process packets for the device by adding the it to a poll list. When the corresponding softirq kernel thread is scheduled, it executes the function `net_rx_action` [4].

This function, executed in the context a softirq, processes all devices in the poll list. During the processing of the RX_Ring of a network device, the hardware interrupts for this device are deactivated. Each packet is wrapped in an `sk_buff` structure and handled by an appropriate Layer 3 protocol handler. In the case of IP packets, the `ip_rcv()` function is used. The process is repeated until there are no frames left in the RX_Ring of the Interface or until a limit, called device weight, is reached. Additionally, a new descriptors are allocated and added to the RX_Ring [4, 29, 15].

The `ip_rcv()` function processes the packets as defined in the protocol, including defragmentation and routing. It then calls the appropriate protocol handler of the layer above. For UDP, this is `udp_rcv()` [31].

During processing by UDP, the system verifies the availability of a socket with the corresponding port number. If available, the packet is copied into the receive buffer of the socket. If no corresponding socket is found, the packet is dropped [31].

Processing of the packet in the context of the softirq is now complete. The application can retrieve the packet from the receive buffer of the socket using the `receive()` call.

2.4.2.2 Sending a Packet

To send a packet, an application needs an appropriate socket, in the case of UDP packets an Internet socket of type datagram. The `sendto()` function is used to send the data, which contains the actual data as well as the address of the socket to which the data should be sent.

In the Linux kernel, calls to `sendto()` for a UDP socket are handled by the `udp_sendmsg()` function in context of the application. This creates a socket buffer for the packet. Additionally, the UDP packet undergoes initial checks such as the compliance with the maximum length, and the UDP header is generated. Subsequently, the packet is forwarded to the IP protocol handler [31].

The Internet Protocol handler generates the IP header and fragments the packet if required. In addition, the protocol handler performs routing to determine which network device the packet should be sent to. This process is also performed in the context of the application [31, 4].

To implement traffic management and prioritization, a layer called queueing discipline (QDisc) is placed between the protocol handler of layer 3 and the device driver. By default, a QDisc called 'pfifo_fast' is used, which is essentially a FIFO queue. The queuing of the packet is also handled in the context of the application [4, 32].

The actual transmission of the packet over the network interface card is usually done in the context of a softirq. The device driver for the interface adds the Ethernet header and places it in the transmission queue of the interface, known as the TX_-Ring. The NIC hardware then fetches the packets from the TX_Ring using DMA and transmits them over the physical medium. An interrupt is generated by the interface to indicate a successful transmission [2, 15].

2.5 Tuning Options

2.5.1 Buffers in the Network Stack

2.5.1.1 Socket Receive Queue Memory

2.5.1.2 RX_Ring

2.5.2 Quality of Service

2.5.3 Offloading

2.5.4 Receive Side Scaling

Receive Side Scaling, or RSS for short, is a technology used to improve network performance. The procedure for receiving a packet is described in 2.4.2.1. The interrupt and softirq described there, which carry out the processing of the received packet, are handled by a single CPU [30].

The concept behind RSS is to distribute network data processing across multiple CPU cores instead of keeping it confined to a single core. Incoming network packets are distributed to different processor cores based on a hash function [26].

For Intel network cards, the hash is calculated based on the packet type. The network interface parses all packet headers and uses specific fields as input values for the hash function. In the case of a non-fragmented UDP packet, the destination and source

IP addresses, along with the destination and source port, are used to calculate a 32-bit hash value. This hash value determines the queue and CPU core for processing the packet. All packets with the same input parameters are considered a related communication flow and have the same hash value. As a result, they are processed by the same CPU [14].

2.5.5 Interrupt Moderation

As explained in 2.4.2, the system generates an interrupt for both incoming and outgoing packets, which can negatively impact performance, particularly at high transmission rates due to the high number of interrupts generated [1]. Interrupt moderation can be used to mitigate this issue by delaying the generation of interrupts until multiple packets have been sent or received, or a timeout has occurred, thereby reducing CPU utilization [2].

The Intel network interface drivers enable the configuration of a fixed timeout value for interrupt moderation or the complete deactivation of interrupt moderation [3]. By default, adaptive interrupt moderation is active, which, according to Intel, provides a balanced approach between low CPU utilization and high performance [3]. The interrupt rate is dynamically set based on the number of packets, packet size, and number of connections. The associated patent [4] provides a detailed description of this process. The interrupt rate is typically set to achieve either low latency or high throughput, depending on the type of traffic. For small datagrams, a low interrupt moderation rate (i.e., a high number of interrupts/s) is selected, while for large datagrams, a high interrupt moderation rate (i.e., a lower number of interrupts/s) is selected.

The interrupt moderation rate of Intel network interfaces can be configured using the 'ethtool' configuration tool within the range of 0 to 235 μs. A value of 0 µs deactivates interrupt moderation. Listing 2.1 shows the configuration of the interrupt moderation rate when sending and receiving on interface 'ethX' to a value of 62 μs.

```
ethtool -C ethX rx-usecs 62 tx-usecs 62
```

Listing 2.1: Configuration of the Interrupt Moderation Rate of the ethX Interface.

3 Method

3.1 Setup

3.1.1 Hardware Setup

3.1.2 Software Setup

3.2 Architecture

3.2.1 Setup with Ethernet Switch

3.2.2 Setup with Star Topology

3.3 TestSuite

3.3.1 Description of Software Design

3.3.2 Generation and Measurement of Target Communication

3.3.3 Recorded and Analyzed Data

3.4 Generation of additional System Load

3.4.1 stress-ng

3.4.2 iPerf

4 Tutorial zu dieser LaTeX-Vorlage

Dieses Kapitel ist spezifisch für die LaTeXVorlage und sollte natürlich in der finalen Abgabe nicht enthalten sein. Dieser Teil der Arbeit kann bei Bedarf durch einen Kommentar einfach ausgeblendet werden (siehe thesis.tex Zeile 125).

4.1 Verwendung dieser Vorlage

Dieses Template ist für die Verwendung mit pdflatex gedacht. Am einfachsten ist es die Vorlage in Overleaf [?] zu öffnen. Overleaf ist eine Onlineanwendung zum Arbeiten mit LaTeX, was den Vorteil hat, dass nichts lokal installiert werden muss und mit jedem Betriebssystem gearbeitet werden kann, das über einen Browser verfügt. Außerdem ist es möglich mit mehreren Personen gemeinsam an einem Projekt zu arbeiten. Die Vorlage kann auch mit einer lokalen Installation verwendet werden. Die Website von Overleaf bietet zudem einige gute Tutorials zum Arbeiten mit LaTeX: `https://www.overleaf.com/learn`.

Fehler und Warnungen, die beim Compilieren erzeugt werden, sollten direkt behoben werden, da es später schwierig sein kann den eigentlichen Auslöser einer Fehlermeldung zu finden. Manchmal sieht das Dokument trotz Fehlermeldung oder Warnung korrekt aus, der Fehler macht sich dann aber später bemerkbar. Die Meldungen sind leider oft nicht sehr aussagekräftig, weshalb es am einfachsten ist direkt nach dem Auftreten eines Fehlers den Teil der Arbeit anzuschauen, der als letztes geändert wurde.

4.2 Projektstruktur

Der Hauptteil einer Thesis besteht üblicherweise aus mehreren Kapiteln, die verschiedene Aspekte der Arbeit beleuchten. Es ist ratsam für jedes Kapitel ein eigene Tex-Datei anzulegen, damit der Quellcode übersichtlich bleibt. Durch einen numerischen Präfix (z.B.: `20_relatedwork.tex` siehe Abb. 4.1) werden die Quelldateien in der richtigen Reihenfolge in Overleaf angezeigt. Wir verwenden 20 anstatt 2, damit wir nachträglich auch noch 21, 22, etc. einfügen können.

4.2.1 Unterkapitel

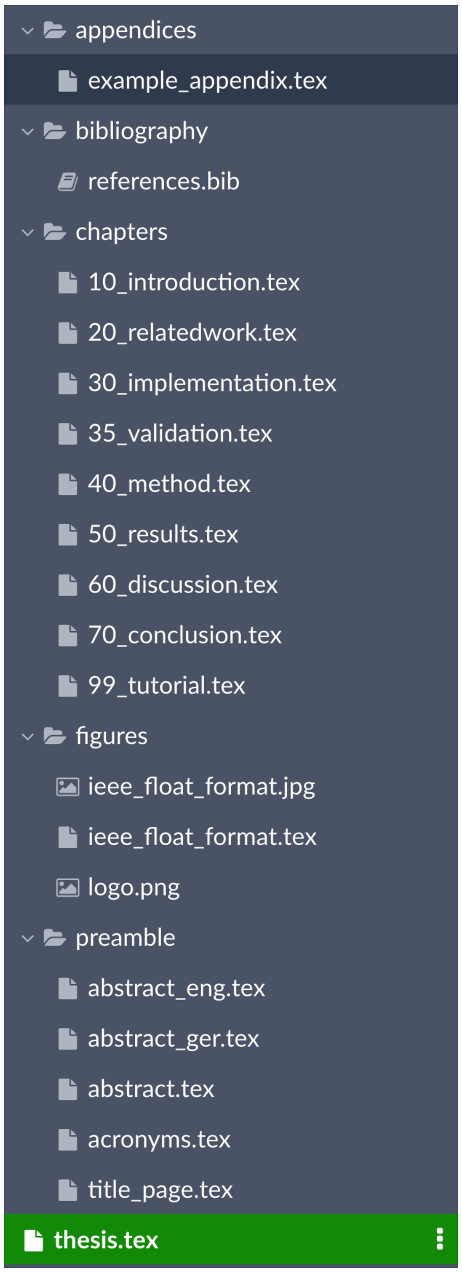
Unterkapitel sollten ein abgeschlossenes Thema behandeln. Einzelne Unterkapitel in einem Kapitel sind zu vermeiden, also z. B. in Kapitel 2 das Unterkapitel 2.1, aber kein weiteres Unterkapitel. In diesem Fall ist es besser entweder den Inhalt von 2.1 direkt in Kapitel 2 zu schreiben, oder falls 2 und 2.1 thematisch zu weit voneinander entfernt sind, aus Unterkapitel 2.1 ein eigenes Kapitel 3 zu machen. Dieses Unterkapitel ist ein negativ Beispiel dafür.

4.3 Grafiken

In der Informatik sind die häufigsten Grafiken entweder Diagramme oder Plots. Beide Arten von Grafiken lassen sich gut als Vektorgrafiken erstellen und einbinden. Der Vorteil von Vektor- gegenüber Pixelgrafiken ist, dass beliebig weit in eine Grafik hereingezoomt werden kann, ohne dass sie unscharf wird. Zudem benötigen Vektorgrafiken meistens weniger Speicherplatz.

4.3.1 Vektor- vs Pixelgrafiken

Für die meisten Abbildungen sollte man Vektorgrafiken verwenden, diese können verlustfrei skaliert werden und bieten somit die meisten Freiheiten. Wenn man Grafiken in R erstellt, können die Plots als pdf oder svg-Dateien abgespeichert werden



```
appendices
    example_appendix.tex
bibliography
    references.bib
chapters
    10_introduction.tex
    20_relatedwork.tex
    30_implementation.tex
    35_validation.tex
    40_method.tex
    50_results.tex
    60_discussion.tex
    70_conclusion.tex
    99_tutorial.tex
figures
    ieee_float_format.jpg
    ieee_float_format.tex
    logo.png
preamble
    abstract_eng.tex
    abstract_ger.tex
    abstract.tex
    acronyms.tex
    title_page.tex
thesis.tex
```

Figure 4.1: Ordnerstruktur eines LaTeX-Projektes

und liegen dann ebenfalls als Vektorgrafik vor. Ein weiteres beliebtes Programm zum Erstellen von Vektorgrafiken ist Inkscape [?]. Zudem bieten viele Programme die Möglichkeit eine Grafik z. B. als PDF zu exportieren, was in LaTeX als Vektorgrafik eingebunden werden kann. Adobe Indesign wird auch häufig zum Erstellen von Vektorgrafiken verwendet.

4.3.1.0.1 Profi-Tipp TikZ.

Grafiken können direkt in LaTeX mit dem TikZ Paket [?] erstellt werden. Die Verwendung ist etwas gewöhnungsbedürftig, da Grafiken mit Code beschrieben werden, bietet aber viele Freiheiten. Außerdem werden die so erstellten Grafiken direkt in LaTeX gerendert und verwenden die selbe Schriftart wie im Text und eine konsistente Schriftgröße im gesamten Dokument.

Pixelgrafiken lassen sich nicht immer vermeiden, z. B. wenn eine Foto in die Arbeit eingebunden werden soll. In diesem Fall sollte darauf geachtet werden, dass die Grafik über eine ausreichende Auflösung verfügt. Eine Auflösung von 300 dpi ist ein guter Richtwert, um beim Drucken ein gutes Ergebnis zu erhalten.

Abbildungen 4.2 und 4.3 zeigen beide den Aufbau des IEEE Floating Point Formats. Abbidlung 4.3 ist eine Pixelgrafik, während Abbildung 4.2 mit TikZ erstellt wurde. Der Unterschied wird beim hereinzoomen deutlich.

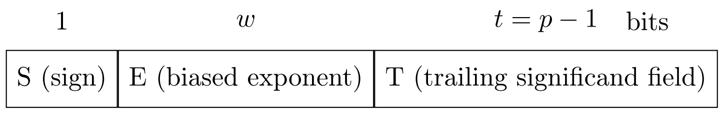


Figure 4.2: Aufbau des IEEE Floating Point Formats als Vektorgrafik mit TikZ erzeugt.

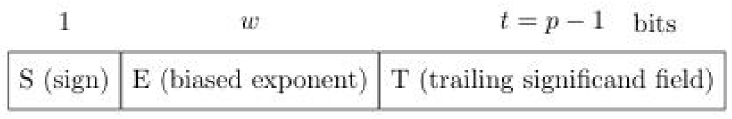


Figure 4.3: Aufbau des IEEE Floating Point Formats als Pixelgrafik.

4.4 Tabellen

Tabellen können in LaTeX direkt erstellt werden. Tabelle 4.1 zeigt ein Beispiel dafür. Einfache Tabellen lassen sich schnell erstellen, bei komplizierteren Tabellen ist es manchmal einfacher zusätzliche Pakete zu verwenden. Mit dem Paket `multirow` können z. B. einfacher Tabellen erstellt werden, bei denen einzelne Zeilen oder Spalten zusammengefasst sind.

Unter `https://www.tablesgenerator.com/` findet man ein hilfreiches online Werkzeug zum erstellen von Tabellen. Wichtig ist es hier den "Default table style" zum "Booktabs table style umzustellen".

Parameter	binary16	binary32	binary64	binary128
k, storage width in bits	16	32	64	128
w, exponent field width in bits	5	8	11	15
t, significand field width in bits	10	23	52	112
emax, maximum exponent e	15	127	1023	16383
bias, $E - e$	15	127	1023	16383

Table 4.1: IEEE 754-2019 Floating Point Formate als Beispiel für das Einbinden einer Tabelle.

4.5 Quellcode

Um Quellcode in die Arbeit einzubinden, können in LaTeX Listings verwendet werden. Es gibt für populäre Sprachen vorgefertigte Umgebungen, welche die Syntax farblich hervorheben. Quellcode sollte eingebunden werden, wenn eine konkrete Implementierung in einer Sprache erläutert wird. Für die Erklärung eines Algorithmus ist es oft übersichtlicher ein Schaubild oder Pseudocode zu verwenden. Es sollten nur kurze Codeabschnitte eingebunden werden, die für den Leser einfach nachvollziehbar sind und nur den für die Erklärung relevanten Code enthalten. Längere Codeabschnitte können im Anhang stehen. Der komplette Code, der für die Arbeit geschrieben wurde, sollte in einem Repository (Gitlab) abgelegt werden.

Listing 4.1 zeigt ein Beispiel für ein Codelisting in der Programmiersprache C.

Algorithmus 4.1 zeigt einen Routing Algorithmus als Pseudocode. Der Code wurde mit dem Paket algorithm2e [?] erstellt.

```c
#include <stdio.h>
// comments are highlighted in green
void main() {
    // keywords of the language are highlighted in blue
    for (int i = 0; i <= 42; ++i) {
        printf("%d\n", i);
    }
    // strings are highlighted in red
    printf("Hello World!");
}
```

Listing 4.1: Beispiel für ein Codelisting in der Sprache C.

```
if destination in west direction then
    go West;
else if destination in same column then
    if destination in north direction then
        go North;
    else
        go South;
    end
else if destination in north east direction then
    go North or go East
else if destination in south east direction then
    go South or go East
else if destination in same row and in east direction then
    go East;
else if at destination then
    done;
```

Algorithmus 4.1: West First-Routing Algorithm.

4.6 Literatur

Ein Literaturverzeichnis sollte mit dem apa Paket für BibLatex erstellt werden. Dazu wird für jede Quelle ein Eintrag in der Datei `references.bib` angelegt. An der passenden Stelle im Text können diese Einträge mit dem `\cite{}` Befehl zitiert werden. Für jede Quelle die zitiert wird, legt LaTeX im Literaturverzeichnis einen Eintrag an.

Beschreibungen der Quellen im Bibtex-Format müssen meistens nicht selbst erstellt werden, sondern können direkt bei vielen Verlagen und Bibliotheken direkt generiert werden. Google bietet mit dem "Scholar-Button" ein Chromium Plugin, mit dem schnell bibtex-Einträge generiert werden können. Bei Google-Scholar generierten Bibtex-Einträgen muss auch eine manuelle Endkontrolle stattfinden, um zu prüfen, ob die Daten in Google korrekt gespeichert waren (z.B. fehlende Autoren, falsche Jahreszahl, etc.).

Hier[1] gibt es ein hilfreiches Cheat-Sheet.

Hinweis: Keine dieser Referenzen müssen Sie in Ihrer Arbeit zitieren!

4.7 Abkürzungen

Für jede verwendete Abkürzung kann ein Eintrag in der Datei `acronyms.tex` angelegt werden. Wenn diese Abkürzung im Text zum ersten Mal auftaucht, sollte der Begriff ausgeschrieben werden mit der Abkürzung in Klammern dahinter. Bei weiteren Vorkommen im Text kann dann die eigentliche Abkürzung verwendet werden. In LaTeX gibt es dafür spezielle Befehle. Beispiel für ausgeschriebene Abkürzung (siehe Quelltext des Dokuments für die entsprechenden Befehle).

Erste Verwendung mit Erläuterung: Network Interface Controller (NIC).
Beispiel für das Verwenden der Abkürzung: NIC.

Die verwendeten Abkürzungen werden automatisch im Abkürzungsverzeichnis aufgelistet.

[1] `https://tug.ctan.org/info/biblatex-cheatsheet/biblatex-cheatsheet.pdf`

´

5 Bibliography

[1] ISO/IEC 11801-1:2017: Information technology - Generic cabling for customer premises, Part 1: General requirements. International Standard, 2017.

[2] Christian Benvenuti. *Understanding Linux Network Internals.* O'Reilly Media, Inc., 2005.

[3] Daniel P. Bovet and Marco Cesati. *Understanding the Linux Kernel.* O'Reilly Media, Inc., 3 edition, 2006.

[4] Ashwin Chimata. Path of a packet in the linux kernel stack. 07 2005.

[5] Jonathan Corbet, Alessandro Rubini, and Greg Kroah-Hartman. *Linux Device Drivers.* O'Reilly Media, Inc., 3 edition, 2005.

[6] Elektronik-Kompendium. 10-Gigabit-Ethernet / 10GE / IEEE 802.3ae / IEEE 802.3an, 2023. Accessed on 27.11.2023. URL: `https://www.elektronik-kompendium.de/sites/net/1107311.htm`.

[7] Elprocus. Pulse Amplitude Modulation, 2023. Accessed on 27.11.2023. URL: `https://www.elprocus.com/pulse-amplitude-modulation/`.

[8] Fachhochschule München, Fachbereich Elektrotechnik und Informationstechnik. Sockets in LINUX – Grundlagen, 2023. Accessed on 26.12.2023. URL: `https://www-lms.ee.hm.edu/~seck/AlleDateien/VERTSYS/Vorlesung/UebersichtSocket1.pdf`.

[9] IEEE P802.3dj Task Force. IEEE 802.3 Ethernet WG Opening Plenary, 2023. Accessed on 27.11.2023. URL: `https://grouper.ieee.org/groups/802/3/minutes/mar23/2303_3dj_open_report.pdf`.

[10] Eduard Glatz. *Betriebssysteme: Grundlagen, Konzepte, Systemprogrammierung.* Dpunkt.Verlag GmbH, 2019.

[11] Jens Heuschkel, Tobias Hofmann, Thorsten Hollstein, and Joel Kuepper. Introduction to raw-sockets. Technical Report TUD-CS-2017-0111, Technische Universität Darmstadt, 2017.

[12] Heiko Holtkamp. TCP/IP im Detail: Internet-Schicht, 2001. Accessed on 21.11.2023. URL: `http://www.cfd.tu-berlin.de/Lehre/EDV2/tcpip/kap_2_3.html`.

[13] Heiko Holtkamp. TCP/IP im Detail: Transportschicht, 2001. Accessed on 21.11.2023. URL: `http://www.cfd.tu-berlin.de/Lehre/EDV2/tcpip/kap_2_4.html`.

[14] Intel Corporation. Intel Ethernet Controller 700 Series: Hash and Flow Director Filters, 2018. Accessed on 17.12.2023. URL: `https://www.intel.com/content/www/us/en/developer/articles/technical/intel-ethernet-controller-700-series-hash-and-flow-director-filters.html`.

[15] Intel Corporation. *Intel Ethernet Controller X710/XXV710/XL710 Datasheet*, 06 2022. Rev. 4.1.

[16] International Business Machines Corporation. Socket Programming on IBM i 7.4, 2018. Accessed on 26.12.2023. URL: `https://www.ibm.com/docs/en/ssw_ibm_i_74/rzab6/rzab6pdf.pdf`.

[17] Steven Iveson. IP Fragmentation in Detail, 2019. Accessed on 29.11.2023. URL: `https://packetpushers.net/ip-fragmentation-in-detail/`.

[18] Kernel Development Community. Networking - The Linux Kernel Documentation, 2023. Accessed on 27.12.2023. URL: `https://docs.kernel.org/networking/index.html`.

[19] Michael Kerrisk. *The Linux Programming Interface: A Linux and UNIX System Programming Handbook.* No Starch Press, 2010.

[20] Linux Manual Page Contributors. packet(7) - Linux Manual Page. Man7.org, 2023. Accessed on 27.12.2023. URL: https://man7.org/linux/man-pages/man7/packet.7.html.

[21] Linux Manual Page Contributors. raw(7) - Linux Manual Page. Man7.org, 2023. Accessed on 27.12.2023. URL: https://man7.org/linux/man-pages/man7/raw.7.html.

[22] linux.conf.au. So you're a linux kernel developer? Name all subsystems, 2021. Video, Accessed on 11.12.2023. URL: https://www.youtube.com/watch?v=YDNzKGTl_PY.

[23] Robert Love. *Linux Kernel Development.* Addison-Wesley Professional, 3 edition, 2010.

[24] Stefan Luber and Andreas Donner. Definition: Was ist 10GbE?, 2018. Accessed on 27.11.2023. URL: https://www.ip-insider.de/was-ist-10gbe-a-680925/.

[25] Ullrich Margull. Betriebssysteme. Script for the course "Betriebssysteme", 2020. Technische Hochschule Ingolstadt.

[26] Microsoft Corporation. Introduction to Receive Side Scaling, 2023. Accessed on 17.12.2023. URL: https://learn.microsoft.com/en-us/windows-hardware/drivers/network/introduction-to-receive-side-scaling.

[27] Shubham Negi. DCSE252 — Course Notes — Unit 1–5, 2023. Accessed on 29.11.2023. URL: https://medium.com/@shubham64negi/cn-unit-1-5-9b60cbf94230.

[28] Red Hat. What is the Linux Kernel?, 2019. Accessed on 10.12.2023. URL: https://www.redhat.com/en/topics/linux/what-is-the-linux-kernel.

[29] Red Hat, Inc. Overview of Packet Reception - Red Hat Enterprise Linux 6, 2023. Accessed on 29.12.2023. URL: https://access.redhat.com/documentation/de-de/red_hat_enterprise_linux/6/html/performance_tuning_guide/s-network-packet-reception.

[30] Red Hat, Inc. Receive-Side Scaling (RSS) - Red Hat Enterprise Linux 6, 2023. Accessed on 17.12.2023. URL: https://access.redhat.com/documentation/de-de/red_hat_enterprise_linux/6/html/performance_tuning_guide/network-rss.

[31] Rami Rosen. *Linux Kernel Networking: Implementation and Theory*. Apress, 2013.

[32] Dan Siemon. Queueing in the Linux Network Stack, 2013. Accessed on 29.12.2023. URL: https://www.linuxjournal.com/content/queueing-linux-network-stack.

[33] Andrew S. Tanenbaum and David J. Wetherall. *Computer Networks*. Prentice Hall Press, USA, 5th edition, 2010.

[34] The Linux Information Project. Kernel Control Path Definition, 2006. Accessed on 10.12.2023. URL: https://www.linfo.org/kernel_control_path.html.

[35] Paul S. Wang. *Mastering Modern Linux*. Chapman & Hall, 2 edition, 2018.

[36] Inge Weigel. Computer Networks. Lecture Material in the Course "Computer Networks", 2021. Technische Hochschule Ingolstadt.

[37] Robert Winter, Rich Hernandez, Gaurav Chawla, et al. Ethernet jumbo frames. Technical report, Ethernet Alliance, Beaverton, OR, November 2009. URL: http://www.ethernetalliance.org/wp-content/uploads/2011/10/EA-Ethernet-Jumbo-Frames-v0-1.pdf.

A Appendix

Der Anhang kann Teile der Arbeit enthalten, die im Hauptteil zu weit führen würden, aber trotzdem für manche Leser interessant sein könnten. Das können z. B. die Ergebnisse weiterer Messungen sein, die im Hauptteil nicht betrachtet werden aber trotzdem durchgeführt wurden. Es ist ebenfalls möglich längere Codeabschnitte anzuhängen. Jedoch sollte der Anhang kein Ersatz für ein Repository sein und nicht einfach den gesamten Code enthalten.

B List of Figures

C List of Tables

D Listings