Contents

1 Introduction

1.1 Motivation

Motivation – Warum soll dies untersucht werden? Kurze Einordnung. Vorstellung TSS (kompakt).

1.2 Related Work

Vorstellung von 3 Papern die auch UDP Kommunikation untersuchen, Heraustellung der Besonderheiten in dieser Arbeit

1.3 Research Questions

3 Forschungsfragen aus Themenbeschreibung, mit Requirements an Arbeit (evtl. als seperaten Punkt).

2 Background

2.1 Positioning in the distributed Test Support System

TODO – Beschreibung des Senden/Empfangen im verteilten TSS zur besseren Einordnung der Arbeit

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 [79]. 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 [84].

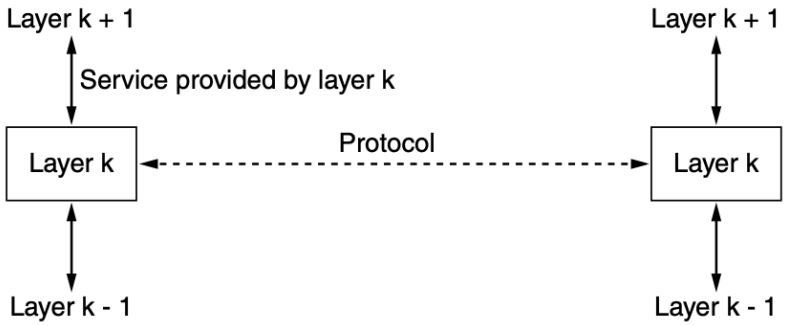

Figure 2.1: Relationship between service and protocol. Source: [79].

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 [79].

A protocol is a set of rules that define the format of messages exchanged within a layer [79]. 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 [84].

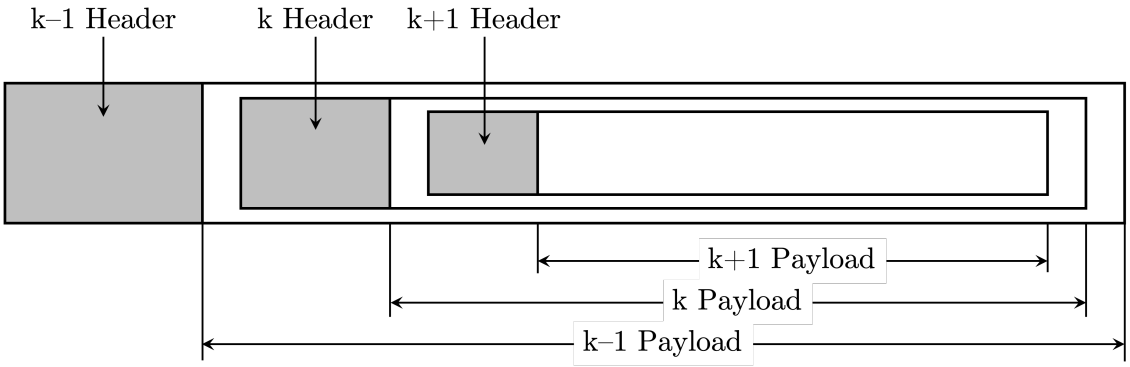

Figure 2.2: Encapsulation Principle. Adapted from: [79].

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 [79].

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 [79]. 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 [79].

- The **Physical Layer** serves as the interface between a network node and the

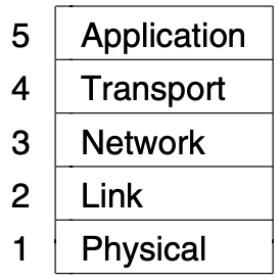

Figure 2.3: Hybrid TCP/IP Reference Model. Source: [79].

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 [79, 84].

- 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 [84].
- The **Network Layer** provides end-to-end communication between two network nodes. This includes addressing and routing [79, 84].
- 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 [84].
- 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 [84].

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.

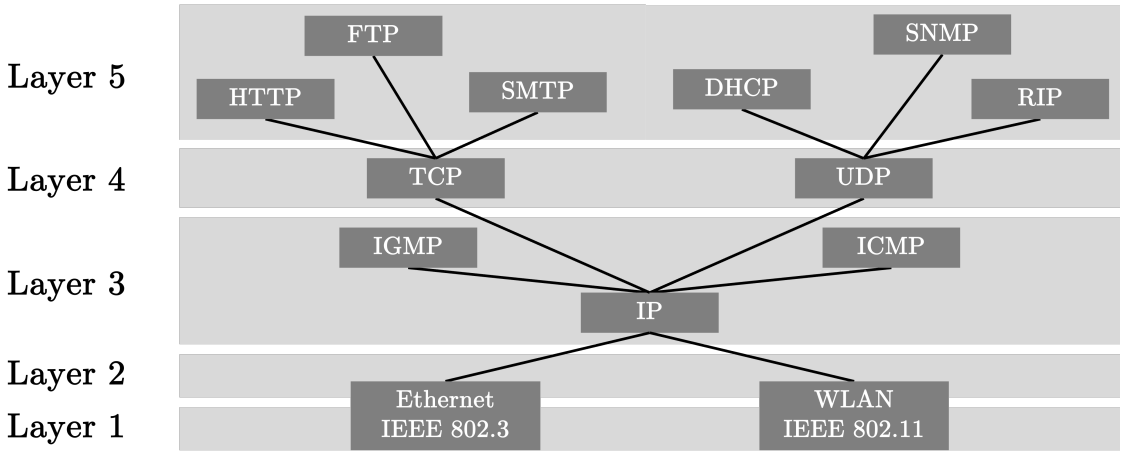

Figure 2.4: Selection of important protocols of the hybrid TCP/IP Reference Model. Adapted from: [84].

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 [79]. Further information about the protocols of the TCP/IP reference model can be found in [79].

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 [19]. 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 [16], 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 [17]. 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 [16].

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.

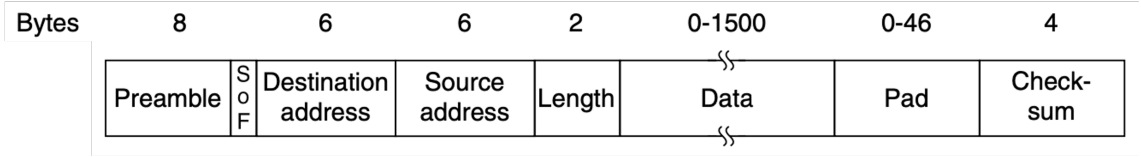

Figure 2.5: Structure of the Ethernet frame. Source: [79].

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 [79].

This is followed by the *destination* and *source address*. This is the MAC address, which is uniquely assigned globally to a network interface [84]. 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 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 [85]. Jumbo frames can increase network throughput and reduce CPU usage, as demonstrated in studies [68]. It is important to ensure that all network participants support jumbo frames to avoid packet loss [27].

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 [79]. 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 [79].

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 [60].

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 [84]. 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.

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

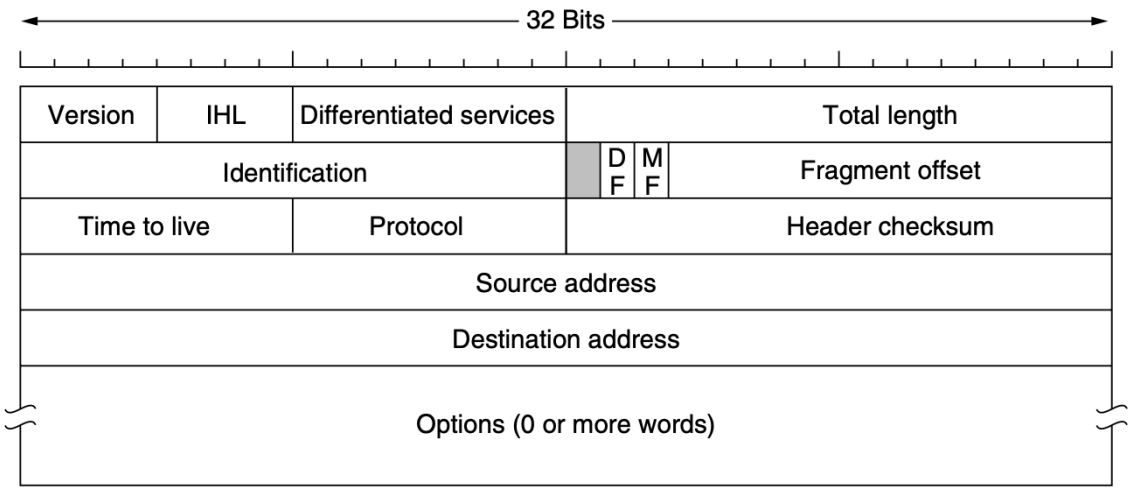

Figure 2.6: Structure of the IP Header. Source: [79].

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 [84], 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 [24]. 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 [79].

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

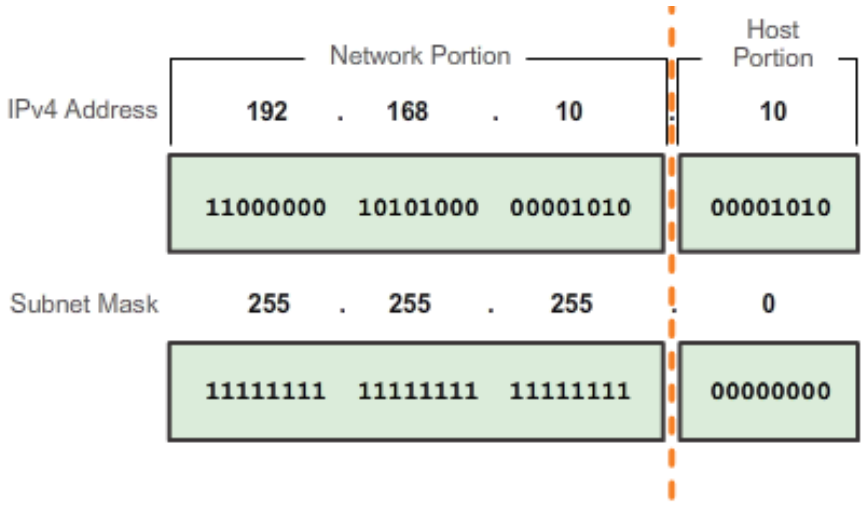

Figure 2.7: Structure of the IP address and subnet mask. Source: [66].

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 [84].

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 [37].

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 [25].

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.

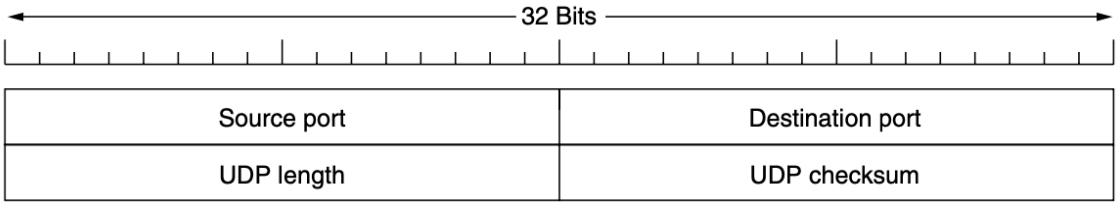

Figure 2.8: Structure of the UDP Header. Source: [79].

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.

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 [83]. 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 [5], [42] and [59], 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 [21].

An operating system kernel serves as the interface between the hardware and the processes of a computer system [69]. It manages hardware resources, schedules processes, and facilitates communication between application software and hardware [61].

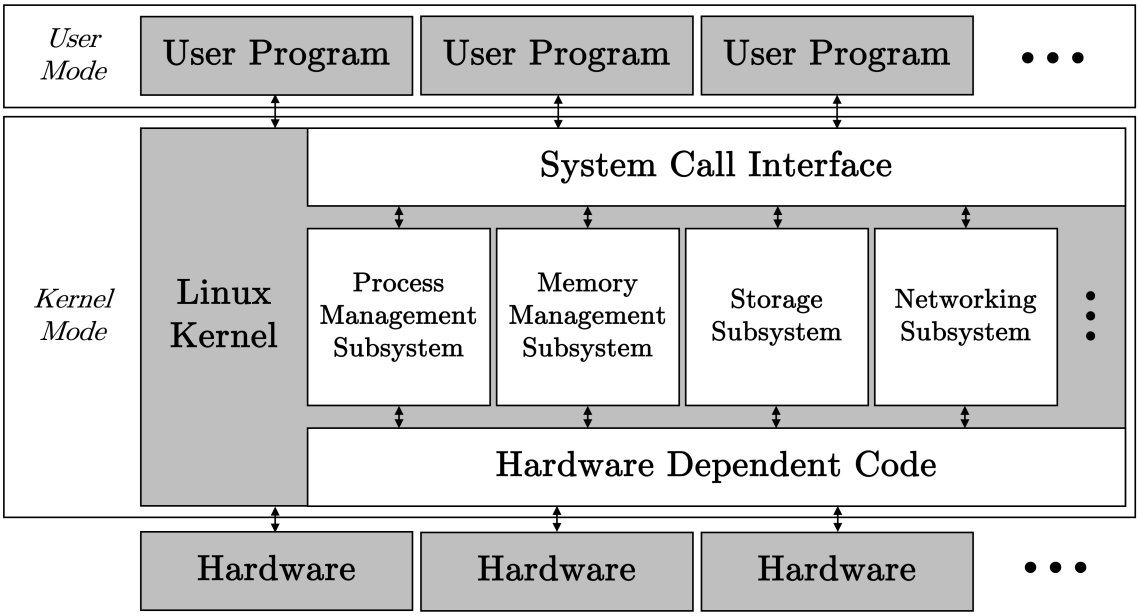

Figure 2.9: Simplified representation of the Linux Kernel with selected Subsystems.

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 [5].

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 [59].

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 [5].

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 [61]. 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 [5]. 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* [80].

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 [5].

It should be noted here that system calls are not the only way to execute instructions in the kernel. According to [5], 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 [12].

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 [12].

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 [21].

Additional information regarding the interface between the Linux kernel and hardware, as well as device drivers, can be found in [12].

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 [58]. Figure 2.9 displays the most significant subsystems, which are further explained below based on [42] and [12].

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.

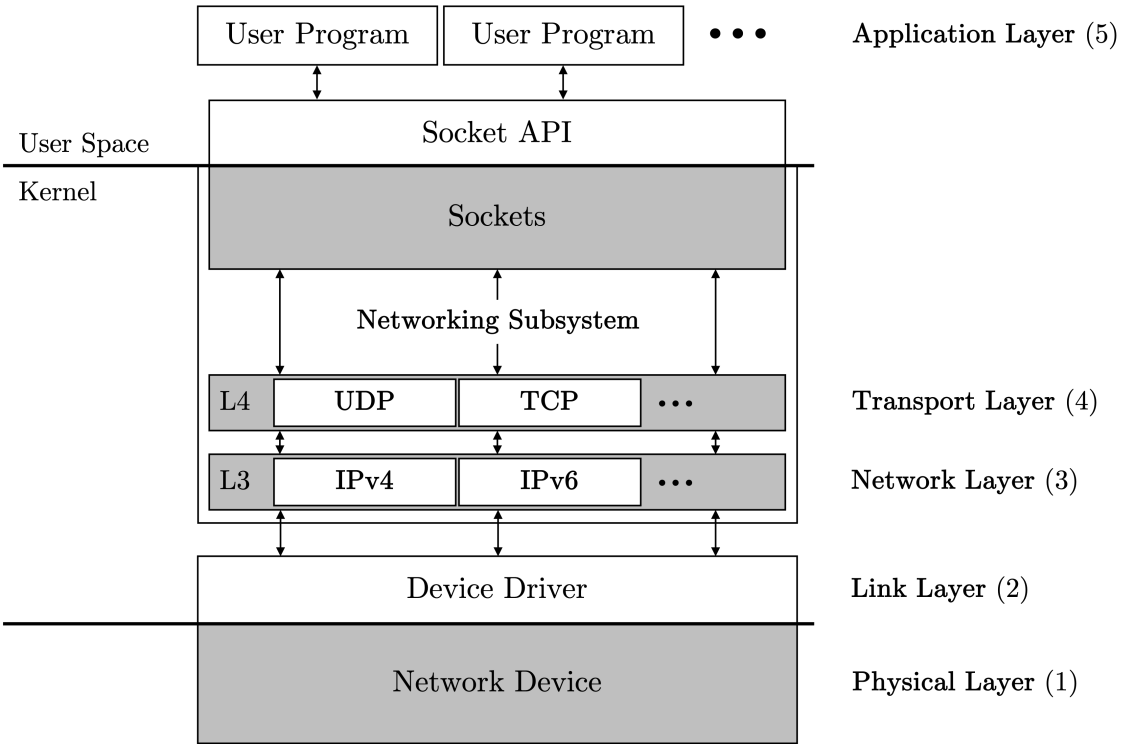

Figure 2.10: Simplified Representation of the Linux Network Stack with associated Layers of the hybrid TCP/IP Reference Model. Adapted from: [3].

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 [3], 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 [18][40].

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 [35].

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 [42].

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 [42].

- **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 [35].

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 [42], 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

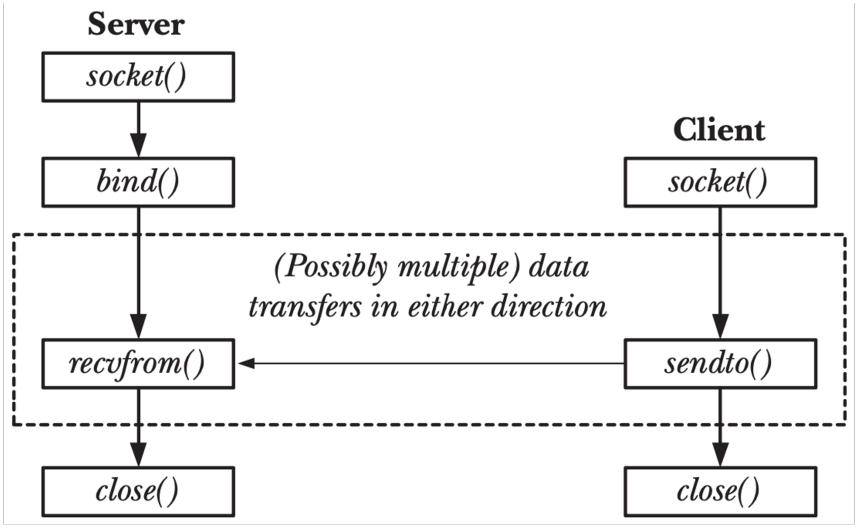

Figure 2.11: Overview of System Calls used with Datagram Sockets. Source: [42].

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 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 [23].

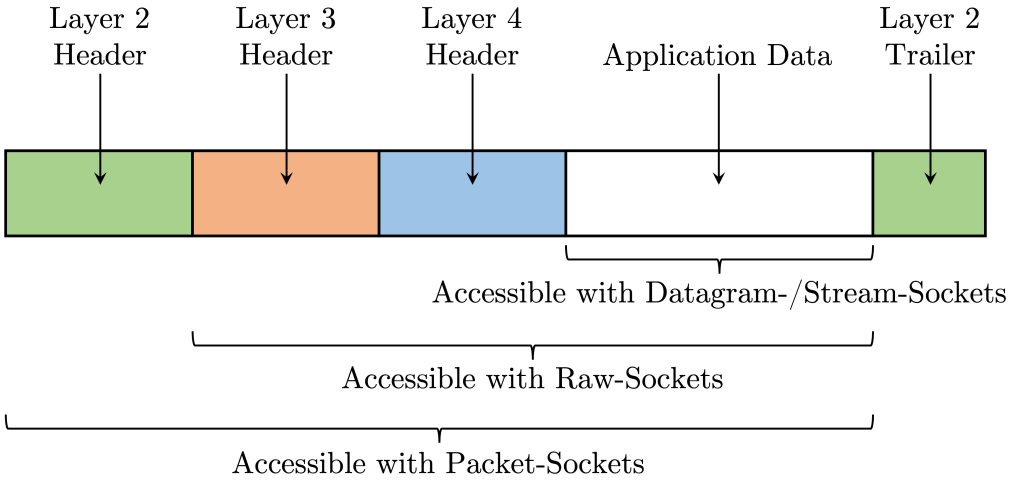

Figure 2.12: Overview of Network Layers and Access Possibilities with different Socket Types. Adapted from: [23].

Figure 2.12 displays the access possibilities with different socket types. Raw sockets provide access to the transport layer (4) and network layer (3) of the network stack [56], 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 [55].

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 [56]. 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' [23] 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 [3].

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 [3].

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 [76].

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 [3].

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 [76]):

- 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 [76].

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 [76]. 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 [7, 29].

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 [76] and [3].

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 [7, 3, 29].

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 [3].

To minimize processing in the interrupt context, softirqs are utilized [7]. 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 [5].

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` [7].

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 [7, 71, 29].

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()` [76].

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 [76].

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 [76].

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 [76, 7].

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 [7, 77].

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 [3, 29].

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 [73].

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 [63].

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 [26].

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 [28]. 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 [62].

The Intel network interface drivers enable the configuration of a fixed timeout value for interrupt moderation or the complete deactivation of interrupt moderation. By default, adaptive interrupt moderation is active, which, according to Intel, provides a balanced approach between low CPU utilization and high performance [34]. The interrupt rate is dynamically set based on the number of packets, packet size, and number of connections. The associated patent [47] 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.

3 Methodology

3.1 Setup

3.1.1 Hardware Setup

3.1.1.1 Computer Systems

The test setup consisted of five computer systems, which can be classified into three types. Specifically, there were two 'High-Performance PCs' (HPC1 and HPC2), two 'Traffic PCs' (TPC1 and TPC2), and one system from the Concurrent iHawk platform. The hardware was intentionally chosen to represent different performance classes in order to identify possible limitations during the tests.

3.1.1.1.1 Hardware of the Computer System Types

Table 3.1 provides an overview of the hardware of the computer system types used. The Hardeare was intentionally chosen to represent different performance classes.

Table 3.1a shows that the High-Performance PCs use an Intel Core i9 13900 CPU with Intel Hybrid Technology, providing 8 Performance-Cores and 16 Efficient-Cores [30]. However, due to incompatibility with the operating system, the efficient cores are disabled, resulting in the use of 8 physical cores or 16 logical cores for systems of this type.

Category	Hardware
CPU	Intel Core i9 13900
RAM	32 GB DDR5 6400 MHz
Mainbaord	ASUS PRIME Z790-P WIFI
Disk	2 TB NVMe M.2 SSD

(a) High-Performance PC

Category	Hardware
CPU	Intel Core i7-3770S
RAM	16 GB DDR3 1333 MHz
Mainbaord	GA-Z77X-UD5H (TPC1), GA-Z77X-UD3H (TPC2)
Disk	256 GB SATA III SSD

(b) Traffic PC

Category	Hardware
CPU	**2x** Intel Xeon Gold 6234
RAM	48 GB DDR4 2400 MHz
Mainbaord	Supermicro X11-DPi-N
Disk	2 TB HDD

(c) iHawk

Table 3.1: Overview of the Hardware of the Computer Systen Types

3.1.1.1.2 Comparison with Computer Systems in the Test Support System

When selecting the hardware, care was taken to ensure that it was similar or identical to the hardware used in a distributed Test Support System.

3.1.1.1.2.1 Systems of the Type 'Traffic PC'

The 'Traffic PC' systems used in this context are similar to the I/O PCs used in the distributed test system. Both systems use the CompactPCI Serial architecture, which allows for modular systems consisting of a system module with the CPU and up to eight peripheral modules. These modules are connected to the system module through serial point-to-point connections [67].

The SC5-FESTIVAL card manufactured by EKF Elektronik GmbH serves as the system module in the distributed test system. It is equipped with an Intel Core i3 7100E processor [15], which provides comparable performance to the Intel Core i7-3770S used in the Traffic PCs [82].

3.1.1.1.2.2 iHawk Platform

iHawk is a computer platform manufactured by Concurrent that is designed with a focus on time-critical simulation or data acquisition [10]. The system used for the tests in this thesis, with the data described in Table 3.1c, will also be used with the same configuration in a distributed test system. This ensures that the conditions of the tests carried out here are comparable to those of the distributed test system.

3.1.1.1.3 Characteristics of the used iHawk System

In the following, the special features of the iHawk system used, which were taken into account in the analyses carried out with special test scenarios, will be discussed.

The iHawk is a dual-socket system, as shown in the block diagram (Figure 3.1). It utilizes a NUMA (Non-Uniform Memory Access) memory design, which means that memory performance varies at different points in the address space, depending on whether it is local memory or the memory of the other processor. Typically, access to local memory is significantly faster than to the memory of the other processor. A CPU with its local memory is referred to as a NUMA node [45].

To access the memory of the other NUMA node, an interconnect between the sockets is used. This can lead to potential problems such as:

- Increase in latency for memory access
- Throughput bottleneck

The iHawk system utilizes two Intel Ultra Path Interconnect (UPI) links, as illustrated in Figure 3.1. Each link operates at a speed of 10.4 GT/s, providing a total full-duplex bandwidth of 41.6 GB/s [65]. Accessing memory from the other NUMA node via the UPI increases latency by approximately 50% [45], resulting in a memory latency of around 130 ns [64] between the two sockets.

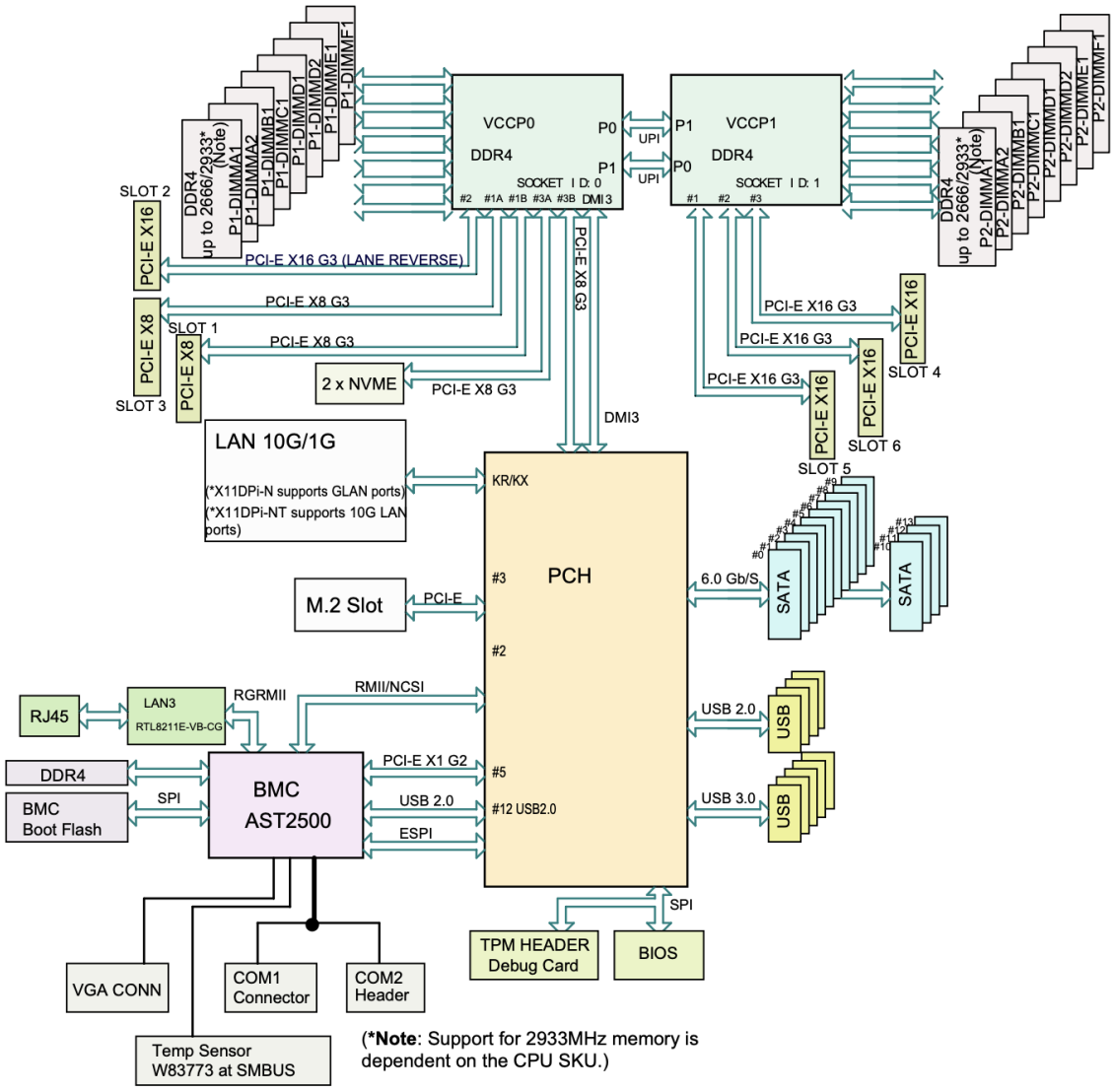

Figure 3.1: Block Diagram of the Supermicro X11-DPi-N Mainboard. Source: [78].

The block diagram shows that each PCI Express slot is connected to one CPU. Slots 0 to 3 are connected to CPU 0, while slots 4 to 6 are connected to CPU 1. It is important to note that the problems previously described for memory accesses also apply to accessing PCI Express devices on the other NUMA node, as they are also carried out via the UPI links.

3.1.1.2 Network Hardware

3.1.1.2.1 Ethernet Switch

The Ethernet switch used is a Cisco CBS350-8XT from the Cisco Business 350 series. It is a Layer 3 managed switch that supports 10 GbE. Its specifications are as follows [8]:

- 8x RJ45 10 GbE Ports
- 2x SPF+ (shared with RJ45 Port)
- 160 GBit/s switching capacity
- 6 MB packet buffer dynamically shared across all ports
- Quality of Service (QoS) with 8 hardware queues
- Support for jumbo frames with a maximum size of 9000 bytes

The Cisco CBS350-8XT is a Layer 3 switch that can forward packets based on both MAC and IP addresses, similar to a router. However, for the purposes of the tests conducted in this thesis, this functionality was not required, so the switch was configured as a Layer 2 switch.

3.1.1.2.2 Network Interface Cards

The tests only use PCIe Network Interface Cards (NIC) that are capable of 10 GbE. The main types of network interfaces used are Intel's X520-DA2, X540-T2, and X710-T2L. Network interfaces from Lenovo and Inspur are also considered. Table 3.2 provides a concise overview of the network interfaces used according to [31, 32, 33, 46].

The network interfaces were selected broadly to reduce dependence on specific interfaces or manufacturers. Additionally, network cards of varying ages and prices were considered to account for potential hardware limitations.

Chapter 3.2 explains for each topology which network interfaces are used on which

	Intel X520-DA2	Intel X540-T2	Intel X710-T2L	Inspur X540-T2	Lenovo ThinkSystem Marvell QL41134
Launch Year	2009	2012	2019	unknown	
Port Configuration	Dual (SPF+)	Dual (RJ45)	Dual (RJ45)	Dual (RJ45)	4 (RJ45)
System Interface	PCIe v2.0 (5 GT/s), x8 Lane	PCIe v2.1 (5 GT/s), x8 Lane	PCIe 3.0 (8.0 GT/s), x8 Lane	CIe v2.1 (5 GT/s), x8 Lane	PCIe 3.0 (8.0 GT/s), x8 Lane
Controller	Intel 82599	Intel X540	Intel X710-AT2	Intel X540-AT2	QLogic QL41134
Data Rate	max. 10 GbE	max. 10 GbE	max. 10 GbE	max. 10 GbE	max. 10 GbE
Offloading Mechanism	GRO, Fragm. Offload	GRO, Frag. Offload	GRO, Frag. Offload	GRO, Frag. Offload	GRO, Frag. Offload

Figure 3.2: TODO - THIS SHOULD BE UPDATED TO A TABLE

computer system. It should be noted that the Intel X710-T2L network interfaces are not compatible with computer systems of the 'Traffic PC' type.

3.1.1.2.2.1 Comparison with Network Interfaces in the Test Support System

In the distributed test system, for systems similar to the 'High Performance PCs' and the iHawk, a wide range of PCIe 10GbE network interfaces can be used. However, the CompactPCI Serial systems can only use network interfaces for which a corresponding peripheral module is available.

The Distributed Test System utilizes the SN5-TOMBACK peripheral module from EKF [14]. This module uses the Intel 82599 controller, has two SPF+ ports and supports 10 GbE. It is therefore comparable to the Intel X520-DA2 network card in the test setup, which uses the same controller.

3.1.1.2.3 Cabling

The cabling of the hardware used was carried out using Cat7 Ethernet patch cables with RJ45 plugs or with fiber optic cables and Intel SPF+ SR modules. Both cabling systems used are suitable for 10 GbE.

3.1.2 Software Setup

This chapter describes the software versions used and important configurations.

3.1.2.1 Versions

3.1.2.1.1 Operating System

The operating system used on all computer systems is the real-time operating system RedHawk Linux 9.2 based on Ubuntu 22.04.3 LTS.

- **Operating system:** RedHawk 9.2 with Ubuntu 22.04.3 LTS user environment
- **Linux kernel:** 6.1.19-rt8-RedHawk-9.2-trace

RedHawk Linux 9.2 is a real-time operating system developed by Concurrent, optimized for real-time determinism with precise and consistent response times and low latency [11]. The manufacturer integrates open source patches and proprietary enhancements into the Linux kernel. The following list briefly describes some important features of the real-time optimized Linux kernel from the product brochure [9]:

- **Standard Linux API**: Since RedHawk is based on a Linux kernel, it offers all standard Linux user level APIs such as POSIX. Therefore, applications created for other Linux distributions can also be executed on the operating system.
- **Frequency-Based Scheduling**: RedHawk has a Frequency-Based Scheduler, which allows processes to be executed in a cyclical execution pattern driven from a real-time clock.
- **Processor Schielding**: RedHawk enables the shielding of individual cores from timers, interrupts, or other Linux tasks, providing a deterministic execution environment.
- **Multithreading and Preemption**: RedHawk allows multiple processes to execute simultaneously in the kernel while protecting critical data or code

sections with semaphores or spinlocks. In the RedHawk kernel, processes can be preemptively interrupted to reallocate CPU control from a lower-priority to a higher-priority process, except during execution in critical kernel sections. To ensure deterministic responses, critical sections of the kernel have been optimized to reduce non-preemptable conditions and enabling high-priority processes to immediately respond to external events, even when the CPU is actively engaged.

RedHawk Linux was chosen for the test setup as it is also used in the Distributed Test System. It also offers low latency, which should have a positive impact on the performance characteristics.

3.1.2.1.2 Drivers of the Network Interface Cards

The drivers supplied with the kernel are used for the network cards. Table 3.2 lists the drivers used for the network cards.

Driver	Network Interface Card
i40e	Intel X710-T2L
ixgbe	Intel X520-DA2, Intel X540-T2, Inspur X540-T2
qede	Lenovo ThinkSystem Marvell QL41134

Table 3.2: Overview of the Drivers of and the associated Network Interface Cards.

3.1.2.2 Configurations

This chapter describes the general configurations that were used for all tests.

3.1.2.2.1 Activation of Jumbo Frames

The tests used jumbo frames with a maximum size of 9000 bytes. Jumbo frames must be supported by all network nodes to avoid packet loss [27]. Since the network in the test setup and in the distributed test system is completely under user control, no problems are expected in this regard.

```
ifconfig ethX mtu 9000
```

Listing 3.1: Configuration of Jumbo Frames for the ethX Interface.

Listing 3.1 shows the configuration of jumbo frames with a maximum size of 9000 bytes for the *ethX* interface.

3.1.2.2.2 Real-time Process

The tests, specifically the test program described in the following section 3.3, were executed as a real-time process with the highest priority. This was done in order to obtain realistic test conditions, as the communication layer is also executed as a real-time process in the distributed Test System.

```
chrt -f -p 99 [PID]
```

Listing 3.2: Modification of the real-time Attributes of a Process.

Listing 3.2 contains the command used to modify the real-time attributes of a process with a specific PID (Process ID). The same command was also applied to the test program. According to [48], the real-time attributes were modified as follows:

- Change of the **scheduling policy** to `SCHED_FIFO`. This is a first in/first out realtime scheduling policy through which the processes have a higher priority than all normal processes and can interrupt them at any time [72].
- Change of the **scheduling priority** to 99. `SCHED_FIFO` uses a fixed priority with values ranging from 1 to 99, with 99 as the highest priority [72].

3.2 Network Topologies

For the tests with the described setup, two different topologies were utilized for the local Ethernet network and the arrangement of the computer systems. Both topologies are star-shaped, consisting of bidirectional point-to-point links that connect two systems. Each link can be used independently in both directions [79].

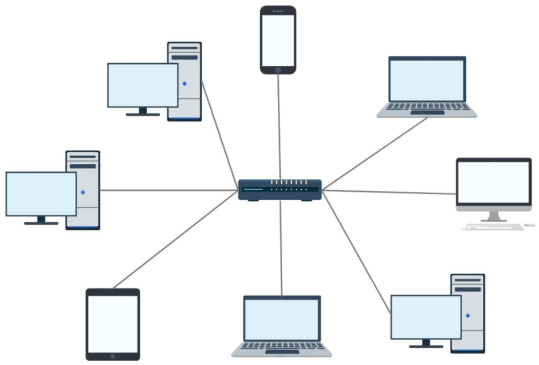

Figure 3.3: Structure of a generic Star Topology. Source: [4].

In a star topology, each node connects to a central instance, typically a hub or a switch. A generic star topology is shown in Figure 3.3. The advantages of the star topology are simple installation and high fault tolerance, as the network remains functional if a node fails. However, if the central instance fails, the network becomes non-functional. Additionally, more cabling is necessary.

Both topologies used are described in detail below. Any modifications made for specific test campaigns are indicated in the corresponding places in the thesis. This includes, for example, changes to the network interfaces used.

3.2.1 Star Topology with a Switch in the Center

The first topology used is a star topology with a switch in the center, as shown in Figure 3.4.

In this architecture, the Cisco CBS350-8XT switch, presented in 3.1.1.2.1, is located in the center of the star. All other participants, including two computer systems of type HPC and the computer systems of type TPC, are connected to this switch.

The HPC1 and HPC2 systems are both equipped with the Intel X710-T2L network card and are each connected to the switch with a bidirectional link using Cat7 copper cables. The TPC1 and TPC2 systems were equipped with the Intel X540-DA2 network card, which has SPF+ ports. They were connected to the switch via fiber optic cables using Intel SPF+ SR transceivers.

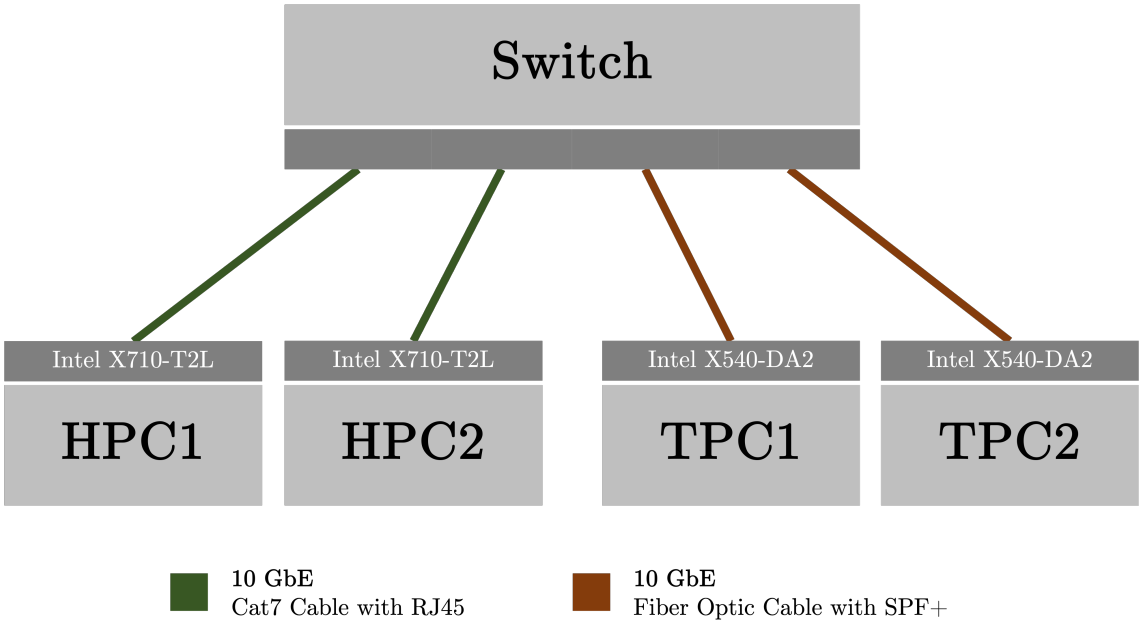

Figure 3.4: Visualization of the Star Topology with a Switch in the Center.

3.2.2 Star Topology with the iHawk in the Center

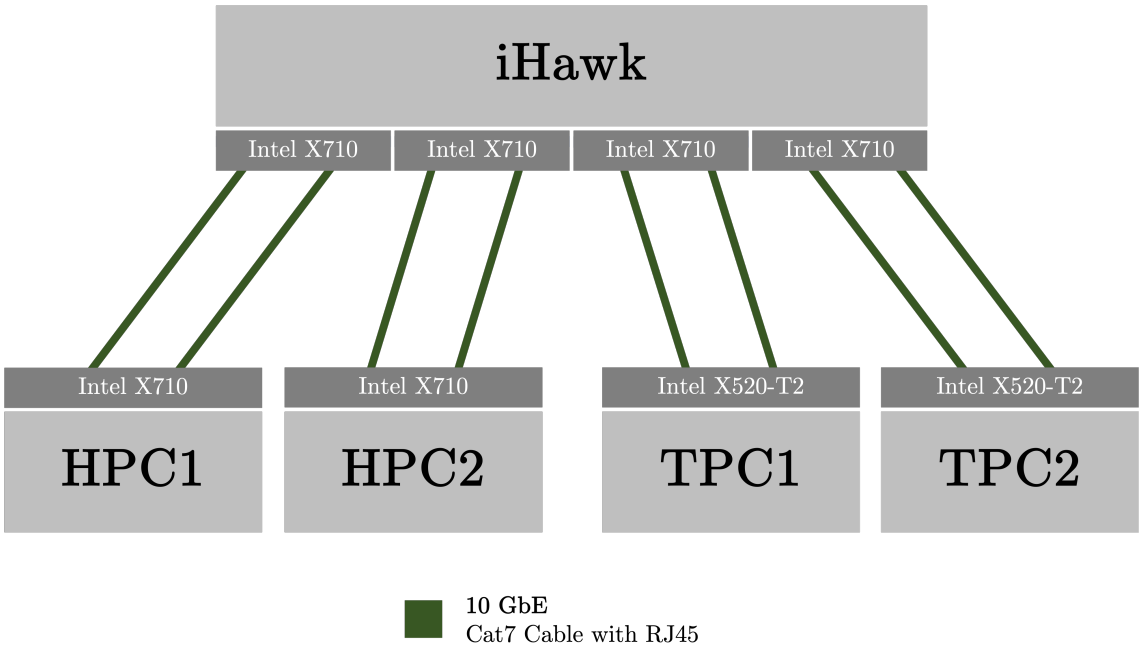

Figure 3.5: Visualization of the Star Topology with the iHawk in the Center.

As the first topology presented proved to be unsuitable in the course of the tests carried out, a new topology as shown in Figure 3.5 was developed.

This new topology features an iHawk computer system at the center of the star, equipped with four Intel X710-T2L network interfaces. The HPC computer systems in this topology also have Intel X710-T2L network interfaces, while the TPC systems use Intel X520-T2 network interfaces. Each node is connected to the iHawk in the center via two bidirectional 10 GbE links.

3.3 Introduction of the Test Program

A dedicated test program, called TestSuite, was created to carry out the tests with the setup described above. The decision to create an own test program was based on the following considerations, which were not fully covered by existing network performance test programs such as iPerf [36]:

- **Execution of tests with UDP protocol**: TestSuite focuses exclusively on tests with the UDP protocol and offers various setting options with regard to the generated and measured UDP communication. In addition to UDP sockets, RAW and PACKET sockets are also supported.
- **Results management**: TestSuite saves all relevant results of a test run in XML files, which facilitates subsequent evaluation.
- **Focus on relevant aspects of the test**: The TestSuite places particular emphasis on testing packet loss and latency. This is reflected in various functionalities that stand out from existing test programs:
  - Recording of losses with intermediate results throughout the test
  - Recording of additional metrics for more precise investigation of packet losses
  - Recording of send and receive timestamps for each packet
- **Automation of tests**: Due to the large number of tests that were expected in

advance, the tests can be automated. This allows a more optimized utilization of the test setup to be achieved. Furthermore, required environmental conditions in the system, such as an additional system load, can be generated automatically.

Disadvantages of creating a custom test program is the additional time required for development and the presence of possible errors that could falsify the results. As the advantages listed above outweighed the disadvantages, the decision was made to develop the TestSuite.

This section first describes the software design of TestSuite. This is followed by a more detailed description of the communication generated and measured by TestSuite. Then, the interpretation and evaluation of the data generated during a test run will be considered and the relevant results recorded by TestSuite will be presented. The complete source code of the TestSuite can be found in the appendix.

3.3.1 Software Design

The TestSuite was developed using the C++ programming language in the C++20 version. The concept of object-oriented programming has been used. The Qt Creator IDE was used for programming. In the following, the concept of the TestSuite is explained before the architecture of the software is explained.

3.3.1.1 Concept

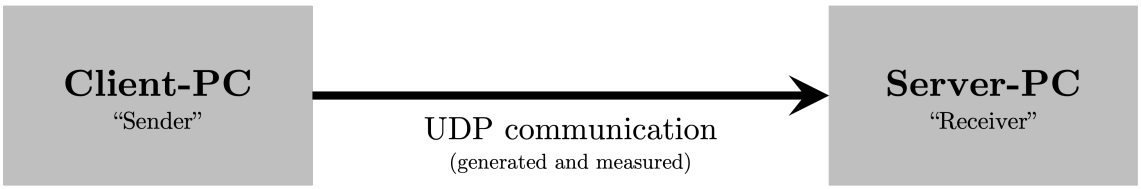

Figure 3.6: Illustration of the TestSuite Concept.

The TestSuite is designed for reliability and performance testing with the topologies presented in 3.2. The basic concept of the TestSuite is shown in Figure 3.6. TestSuite always generates and measures a UDP communication between a sender, also called

a client, and a receiver, also called a server. The focus of TestSuite is on tests performed between two participants in a local network.

TestSuite is able to generate a UDP communication with a pre-defined payload and bandwidth and record associated metrics such as the number of lost packets or the latency between sending and receiving. The generated communication is a one-way communication, i.e. the server does not respond to messages from the client. One execution of TestSuite can generate and measure a maximum of one communication. If more than one communication is to be analyzed in a system, TestSuite can be executed multiple times.

3.3.1.2 Architecture

Figure 3.7 shows the architecture of TestSuite in a simplified form. The central elements are the systems on which TestSuite is executed. These are the client PC and the server PC. The client PC is also responsible for controlling the execution of the tests.

The input and output data is also shown. The diagram also shows the five central classes of the application and their hierarchy, as well as the various communication channels between the client PC and the server PC. The following sections describe these components in more detail.

3.3.1.2.1 Communication Channels

Communication between the client PC and the server PC takes place via three separate communication channels, all using the UDP protocol.

The 'Service Connection' is used to send configuration data to the server. This includes the description of the tests to be performed and the starting and stopping of a test execution. Furthermore, the configurations of the other communication channels, the 'Test Connection' and the 'Feedback Connection', are sent to the server.

The UDP protocol is used for the 'Service Connection'. This is due to the fact that TestSuite focuses on tests between two participants in a local network. However,

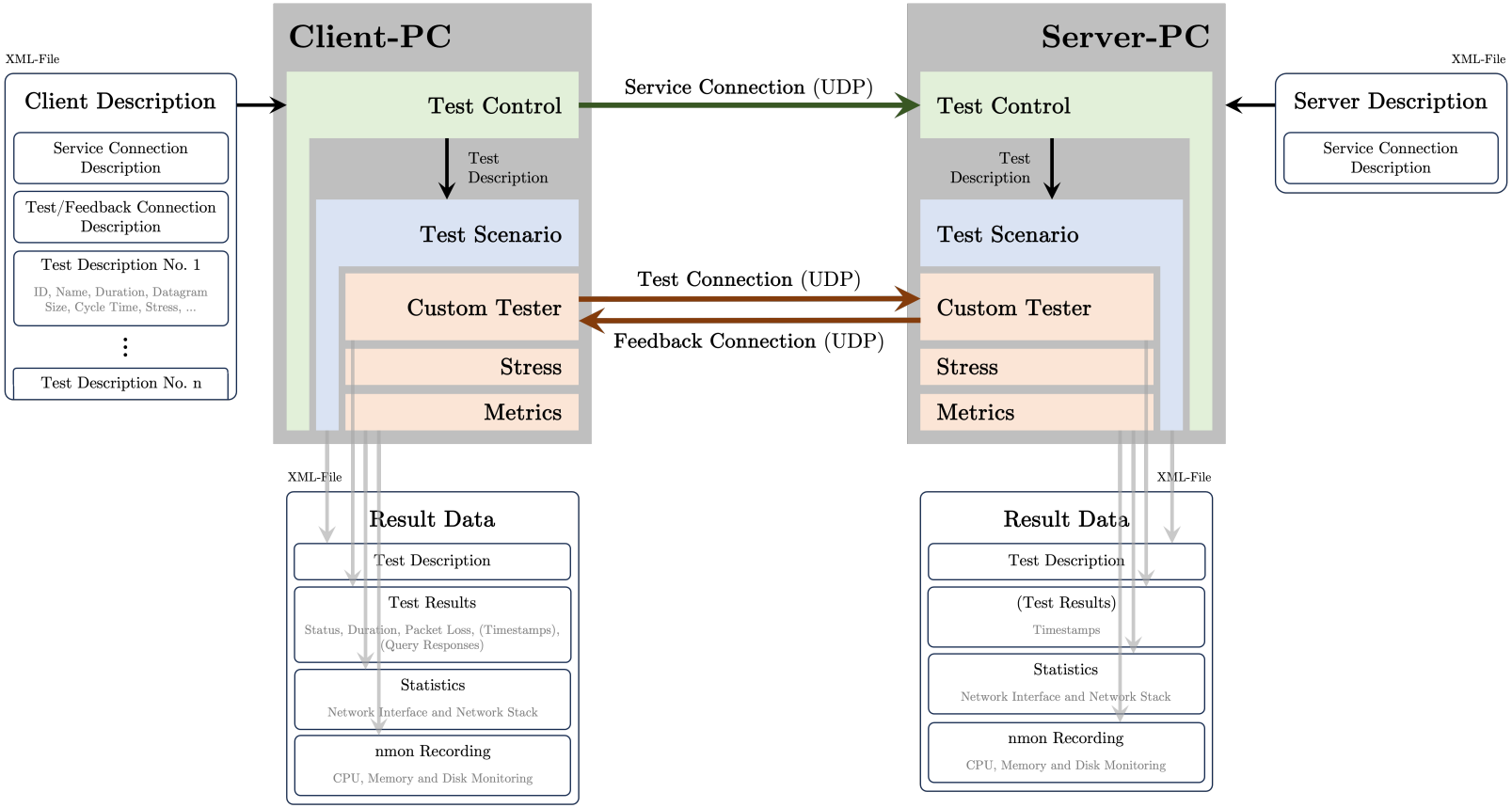

Figure 3.7: Architecture of the TestSuite with the relevant Classes, Data and Connections between the Systems.

UDP does not guarantee that the packets sent will reach their destination. Therefore, if TestSuite is to be used for tests over external networks in the future, a protocol should be used that guarantees the reliability of its transmission.

The 'Test Connection' is the communication channel to be tested. A UDP communication is generated and measured according to certain parameters. This channel exists from the client to the server.

The 'Feedback Connection' allows the server to send messages to the client during the current test run. This is also a UDP connection. The server sends the requested data only when requested by the client.

3.3.1.2.2 Input and Output Data

All data required or created by TestSuite is written in Extensible Markup Language (XML). XML is a markup language for storing structured data in text form [22]. Because XML allows hierarchical storage of data and is human readable, it was used in TestSuite.

The pugixml library was used to read, process and create the XML files in TestSuite. pugixml is a C++ library that allows easy and efficient processing of XML files. More information about pugixml can be found in its documentation [2].

In the following, the input data of the TestSuite, 'Client Description' and 'Server Description', as well as the output data will be examined in more detail.

3.3.1.2.2.1 Input Data

The TestSuite contains all inputs and configurations in the form of a 'Client Description' or 'Server Description'. Both have a similar structure, but the Client Description is more comprehensive because the client is responsible for controlling the tests.

The first element present in both input files is a description of the 'Service Connection'. This is used to exchange test management data between the systems. The description includes the server's IP address and port.

Next, the 'Client Description' contains the configuration of the 'Test Connection' and the 'Feedback Connection'. This includes the IP addresses and port of both channels. In addition, the "Test Connection" description includes the names of the network interfaces used in the client and server, which are required to retrieve statistics. This communication channel also provides the option of using a RAW socket or a PACKET socket in addition to a UDP socket.

The 'Client Description' also contains a description of the tests to be performed, the so-called 'Test Description'. The Client Description can contain any number of Test Descriptions, which are executed in sequence.

The 'Test Description' is a central element of the TestSuite. It contains the complete description of a test execution and must be available in both the client and the server in order to run a test. The 'Test Description' includes:

- **ID** and **name** of the test
- **Path** where the test results are stored
- **Duration** of the test
- Description of the communication to be generated with **datagram size** and **cycle time**
- Description of **stressors** for generating additional system load
- Configurations for **latency measurement** or the use of **query requests**

The data presented here is explained in more detail in 3.3.2 This includes the configurations for generating communication, the generation of additional system load, latency measurement and query requests.

3.3.1.2.2.2 Output Data

Both the Client PC and the Server PC store their output data, called 'Result Data', in the form of XML files. The result data is re-generated for each test.

The first element of the 'Result Data' is the 'Test Description' of the test performed. This makes it easier to associate the output data with a specific test scenario and to

analyze the results.

The output data also includes the test results. These vary depending on the test configuration, but typically include the following elements:

- **Status** of the test execution (success or error)
- **Duration** of the test
- Number of packets **sent** and **received**, with intermediate results from the **query requests** if applicable
- **Timestamp** for the sending or receipt of each packet

The test results in the 'Result Data' for the server are less extensive. They usually only contain the timestamp for the arrival of each packet. The test results and their evaluation are discussed in more detail in Section 3.3.3.

The output data also contains statistics of the network interface or network stack in the respective system. These are recorded before the start and after the end of a test. The statistics are stored:

- **Standard Interface Statistic** of used network interface:
  These statistics include the number of bytes and packets sent and received by the interface. They also include a counter for the number of packets dropped by the interface [39].
- **Network Stack Statistic**:
  These statistics contain information about the protocol layer of the system. The information about UDP and IP is stored in the output data. This includes the number of packets sent and received by each protocol layer, as well as counters for dropped packets. The information about receive errors, such as insufficient buffer space, is particularly relevant for testing.

The output data also includes logs from the nmon tool. This is described in more detail in Section 2.3.X. The data includes minute-by-minute records of the system's CPU usage, memory usage, network interface data, and disk usage information.

3.3.1.2.3 Classes

TestSuite has 5 central classes, the functionality of which is briefly explained below.

3.3.1.2.3.1 Test Control

The 'Test Control' class is responsible for controlling and automating the execution of the test campaign. For this purpose, the class processes the input data of the TestSuite, the 'Client Description' or the 'Server Description'.

An important task is to control the execution of the tests. This is done by the 'Test Control' on the client PC. The tests contained in the Client Description are executed sequentially by creating an instance of the Test Scenario class for each test.

In addition, 'Test Control' in the client sends commands to start and stop a test and the 'Test Description' of the current test to the server via the 'Service Connection'. The server is then able to execute the test accordingly.

3.3.1.2.3.2 Test Scenario

The 'Test Scenario' class handles the overall management of a single test. To create an instance of the class, the 'Test Description' of that test is required as a parameter.

The class will then create instances of the necessary components for that test according to the specifications in the 'Test Description'. This includes:

- Class 'Custom Tester'
- Class 'Stress'
- Class 'Metrics'

3.3.1.2.3.3 Custom Tester

The "Custom Tester" class is responsible for generating and measuring the UDP communication and processing the test results. The generation and measurement of target communication as well as the associated configuration parameters and technical details are described in 3.3.2.

The data measured during a test is then processed and saved as an XML file. "Custom

Tester" saves the "Test Results" section of the "Results Data" output data structure.

3.3.1.2.3.4 Stress

The 'Stress' class is responsible for controlling the generation of additional system load during testing. This includes areas such as CPU load, I/O load, or network load.

The two programs 'stress-ng' and 'iPerf2' are used to generate these loads. These are described in more detail in Section 3.4. The 'Stress' class of TestSuite controls these external programs, including starting, stopping, and configuring them.

The additional system load can be executed during a test on the client PC or the server PC, or on both systems simultaneously.

All configuration data for the additional system load is contained in the Test Description. This includes parameters such as the type of load, its intensity, or its location.

3.3.1.2.3.5 Metrics

The 'Metrics' class is responsible for collecting system metrics before, during, and after a test.

On the one hand, 'Metrics' is responsible for recording the statistics of the network interface or the network stack in the systems before and after a test. The statistics are collected using the command line utilities 'ip' from 'iproute2' and 'netstat'. The output of these utilities is then processed using regular expressions, among other things, and important parameters are stored in the 'Statistics' section of the output data.

Another task of the 'Metrics' class is to control the 'nmon' program. 'nmon' is a system monitoring tool for Linux operating systems. It monitors various system parameters such as CPU usage, memory usage, network traffic, disk activity, and other important system metrics [41]. Furthermore, nmon offers the possibility to record and store the system parameters cyclically.

The TestSuite creates an nmon recording for each test at runtime, in which the

system parameters are recorded every 60 seconds. This recording is part of the 'Result Data'.

3.3.2 Generation and Measurement of Target Communication

As explained in the description of the software design in 3.3.1.2.3.3, the 'Custom Tester' class is used to generate and measure the UDP communication. In the following, this class will be introduced and the send and receive routines will be described in more detail.

3.3.2.1 Parameters and Configuration Options

A UDP communication is generated in the client part of the class and received by the server. Only a one-way communication is considered, i.e. the server does not send any messages other than those required to manage the test run. The generated UDP communication is characterized by the following parameters:

- **Test Duration:** The test duration describes the maximum duration of the test and is specified in seconds.
- **Datagram Size:** As datagram size is referred to the size of the payload of the UDP segment in bytes.
- **Cycle Time:** The cycle time describes the minimum time between two calls of the send function of the socket. This is specified in nanoseconds and realized by a timer. With the cycle time, it is also possible to specify a target bandwidth.

3.3.2.1.1 Query

TestSuite has a query function that can be used to determine the current number of packet losses during the test run.

The client sends a query to the server, which returns the number of packets received so far. The query is synchronized with the test, i.e. no more packets are sent until the

response is received from the server. The query request is made after a pre-defined number of packets have been sent. In the tests performed, a request is sent every 100,000 packets.

The result of the queries, that is the total number of lost packets, is stored in a data structure. This is written to XML data after the test. This makes it possible to view the distribution of packet losses over the duration of the test. The query requests can also be used to abort the test prematurely. If a pre-defined threshold of lost packets, called 'LOSS_THRESHOLD", is exceeded, the test is also terminated before the specified test duration has elapsed.

3.3.2.1.2 Timestamps

The 'Custom Tester' class also supports the recording of timestamps. The client records the time the packet was sent to the application. The server records the time the packet was received in the application. The timestamps are stored on both systems in a data structure that is written to an XML file when the tests are complete.

By calculating the difference between the time stamps, the latency between sending and receiving in the application can be determined. This requires synchronized clocks between client and server.

3.3.2.2 Implementation

Selected features of the TestSuite implementation are described here. This includes the socket abstraction layer as well as the send and receive routine.

3.3.2.2.1 Sockets Abstraction Layer

As mentioned above, the 'Custom Tester' generates a UDP communication. However, this can be generated not only with UDP sockets, but also with raw and packet sockets. To hide the differences in implementation and initialization of the individual socket types, the support class 'uCE' was created, which hides the individual socket

types from the 'Custom Tester' class.

The corresponding socket is initialized in the constructor of the 'uCE' class, which requires the IP address of the client and server and a port number in addition to the socket type. This includes not only the actual creation of the sockets with the Socket API, but also the retrieval of additional information such as the MAC address of the network interfaces of the client and server using an ARP request.

The class provides the send and receive methods for sending and receiving UDP datagrams. Depending on the socket type used, the UDP, IP and Ethernet headers must be generated by the application before sending and removed after sending. This is also done by the class so that the application only receives the payload of the UDP message. For efficiency, no checksum is calculated when the headers are generated.

'uCE' also supports retrieving hardware timestamps from network interfaces to determine the time a packet was received or sent at the network interface. However, this is not supported by the network interfaces used in the test setup.

3.3.2.2.2 Send and Receive Routine

3.3.2.2.2.1 Send Routine

```
int sequence_number = 0;
helpers_timer cycle_timer(CYCLE_TIME);

while(true) {
    if(current_time > end_time)
        break;

    sequence_number++;
    current_time = get_time();
    comm_client.send(TEST_MESSAGE, DATAGRAM_SIZE);

    if(QUERY_ENABLED && ((sequence_number % 100000) == 0)) {
        comm_client.send(QUERY_MESSAGE, sizeof(QUERY_MESSAGE));

        int received_counter = comm_server.receive();
        query_results.push_pack(sequence_number - received_counter)
    ;
```

```
17          if((sequence_number - received_counter) > LOSS_THRESHOLD)
18              break;
19      }
20
21      if (TIMESTAMPS_ENABLED) {
22          timestamps.push_back(current_time);
23      }
24
25      timer_misses += cycle_timer.wait();
26  }
27
28  comm_client.send(STOP_MESSAGE, sizeof(STOP_MESSAGE));
29  int receive_counter = comm_server.receive();
```

Listing 3.3: Simplified Code of the Send Routine.

Listing 3.3 shows a simplified snippet of the send routine. The complete code can be found in the `run()` function of the `custom_tester_client` class in the appendix.

Initializations are done in lines 1 and 2. This includes a counter for the number of packets sent, called 'sequence_number'. In addition, the timer for realizing the cycle time is initialized.

The class 'helpers_timer' is used as the timer. This is also used in other projects in the context of the test support system. The timer implementation ensures a constant time interval between two calls of the `wait` method (see line 25) with the previously defined cycle time. If this interval has not expired, the timer in the `wait` method waits until the interval has expired. Otherwise, it returns immediately. The `wait` method returns the number of timer failures. This is equal to the number of missed time periods between this and the previous call to the method.

This ensures that the time between two calls of the `send` method (see line 10) is at least equal to the previously defined cycle time. It was decided to use the described timer instead of a normal sleep function with a fixed duration to achieve a constant interval between calls to the socket's send method. Since the send method of the Linux socket can block [57], this would not be possible using a sleep function, because the time required to call the `send` method would vary.

Before the start of each transmission loop, the system checks (see line 5) whether the previously specified test duration has been exceeded. If this is the case, the test is terminated. Then, before sending the UDP datagram, the current system time is retrieved from `CLOCK_REALTIME`, the system-wide real-time clock under Linux [49], with a resolution in the nanosecond range (see line 9). This corresponds to the sending time of the datagram in the application.

In line 10 the UDP datagram is sent over the socket type specified in the 'Test Description'. Three different message types have been defined to differentiate the messages in the receiver. In addition to the 'TEST_MESSAGE' used here, there are also the 'QUERY_MESSAGE' and 'STOP_MESSAGE' types, which are described below. For messages of type 'TEST_MESSAGE', the message contains an identifier that identifies the message and the current send counter. The rest of the payload of the UDP datagram is filled with random data to achieve the specified size.

Lines 12 through 19 contain the query request logic. As explained in , a request is sent synchronously to the server, which then sends back its current receive counter. The difference between the send and receive counters corresponds to the current number of lost packets. This is stored in a data structure, namely a vector. If a pre-defined loss threshold is exceeded, the test is terminated prematurely.

In lines 21 to 23, if time stamps are enabled, the previously recorded time stamp is stored in a vector. In addition to the timestamp, the current sequence number is also recorded in order to assign the timestamps. For reasons of readability, this is not included in the simplified code snippet.

At the end of the test, regardless of whether this was triggered by exceeding the previously defined test duration or a loss threshold in connection with query requests, a stop message is sent (see line 28). This stops the receive routine in the server. The receive counter is then sent from the server. This can be used to determine the number of packet losses, which is stored in an XML file along with other metadata such as the exact duration of the test, the number of packets sent, the number of timer failures and, if applicable, the timestamps and results of query requests. This file is supplemented with additional metrics by other TestSuite classes.

The table 3.3 shows the time required for a single iteration of the transmission loop,

Packet Size	Duration for a Loop Iteration
80 Byte	6171 ns
8900 Byte	9726 ns
65000 Byte	63175 ns

Table 3.3: Time for a single Iteration of the Transmission Loop for different Datagram Sizes.

averaged over 1000000 packets. To obtain the minimum times, the timer described above was not used. It can be seen that as the datagram size increases, the time for a loop pass increases. The reason for this is that the call to the Linux API for sending with the corresponding socket increases as the datagram size increases, since more data has to be copied and processed in the kernel [13].

3.3.2.2.2.2 Receive Routine

```
int sequence_number = 0;

while(true) {
    message_type message = comm_client.receive();
    current_time = get_time();

    if(message == TEST_MESSAGE) {
        sequence_number++;

        if (TIMESTAMPS_ENABLED) {
            timestamps.push_back(current_time);
        }
    }
    else if(message == QUERY_MESSAGE) {
        comm_client.send(sequence_number, sizeof(sequence_number));
    }
    else if(message == STOP_MESSAGE) {
        comm_client.send(sequence_number, sizeof(sequence_number));
        break;
    }
}
```

Listing 3.4: Simplified Code of the Receive Routine.

Listing 3.4 shows a simplified snippet of the receive routine. The complete code can be found in the `run()` function of the `custom_tester_server` class in the appendix.

In the receive loop of the server (from line 3), the receive command of the class 'uCE' is called at the beginning, which is blocking. This can be used to determine the message type, shown here in simplified form. Once a packet has been received, the current time is retrieved in the same way as in the send routine.

Then the different types of messages sent by the client are distinguished. If it is a normal test message (type 'TEST_MESSAGE'), the receive counter is incremented. If timestamp recording is enabled, the previously recorded receive time is stored in the vector. Again, the corresponding sequence number is recorded in the real implementation, analogous to the send routine.

If the message is a 'QUERY_MESSAGE', the current receive counter is sent to the client. If the message is a 'STOP_MESSAGE', the current receive counter is also sent. The send loop is then terminated.

At the end of the test, if enabled, the recorded timestamps are written to XML data. As with the client, these are supplemented with additional metrics for the server.

3.3.3 Recorded and Analyzed Data

The presentation of TestSuite has already dealt with the data provided by the program in several places. The following chapter will show how the relevant test results are determined from this data. The most important data provided by the TestSuite are the counters for the number of packets sent and received, a measurement of the exact test duration and the time stamp for each packet. From these, the relevant metrics for reliability and performance testing can be determined.

To calculate the metrics presented below, a Phyton script called "eParser" has been created. This script reads the output data from the TestSuite and calculates the presented metrics and creates diagrams based on them. The script is included in the appendix.

3.3.3.1 Packet Loss

The counters for the sent and received packets can be used to determine the number of packet losses, as shown in equation 3.1. This calculation is already performed by the TestSuite. In the context of the TestSuite and the evaluation of the results, the term "packet loss" is used. Strictly speaking, however, the TestSuite counts the UDP segments sent and received and the calculated number expresses the lost UDP segments.

$$Packet\ Loss = Sent\ Packets - Received\ Packets \tag{3.1}$$

3.3.3.2 Throughput

Throughput is the amount of data transferred per unit of time and can be measured in bits per second. To define throughput, let's consider the example of Host A attempting to send a file to Host B. During the data transfer, two types of throughput can be measured: instantaneous throughput and average throughput. Instantaneous throughput is the rate (in Bits per second) at which Host B receives the file at a given time. Average throughput is defined as the number of bits transmitted divided by the number of seconds it took to complete the transmission [44]. In the context of this thesis, the average throughput as calculated in 3.2 is always referred to.

$$Throughput = (Sent\ Packets \cdot Datagram\ Size)/Duration \tag{3.2}$$

3.3.3.3 Packet Rate

The packet rate (see 3.3) represents the number of packets that are transmitted per unit of time. In contrast to throughput, the packet rate is not specified in bits per second but in packets per second. The average packet rate is considered in this paper.

$$Packet\ Rate = Sent\ Packets/Duration \tag{3.3}$$

3.3.4 Latency

Latency, often referred to as delay, is the amount of time it takes to transmit a message. Latency consists of the following 4 components (according to [20]):

- **Propagation Time:** time required for a signal to travel from the source to the receiver via the transmission path
- **Transmission Time:** time required to transmit all the data bits of a signal from the transmitter source to the receiver
- **Queuing Time:** time that data packets spend in a queue before they are sent
- **Processing Delay:** time required to process data at the sender or receiver, such as adding header information or checking for errors.

TestSuite considers the latency from sending a packet in the client application to receiving a packet in the server application. This can be calculated using the timestamps recorded by the client and server. Thus, the latency can be determined for each transmitted packet.

$$Latency = Receive\ Time - Send\ Time \tag{3.4}$$

To determine the latency as shown in equation 3.4, synchronized clocks between transmitter and receiver are required. The accuracy of the measurement is largely determined by the quality of the clock synchronization.

3.4 Generation of additional System Load

One requirement for the tests is that they should be possible under different operating conditions (see). Based on this requirement, the following categories of load were defined, which should be able to be produced in the test setup, either in isolation or together:

- CPU Load
  In the area of CPU load, the system should be loaded in Linux user mode and kernel mode. The system load should also be generated by real-time processes.
- Memory Load
- I/O Load on the internal hard disk
- Interrupt Load
- Network Load

The existing tools stress-ng and iPerf2 were used to generate the defined load scenarios.

3.4.1 stress-ng

Stress-ng is a system stress testing tool designed to test various aspects of a computer system, such as CPU, memory and I/O, to their performance limits. It is used by system administrators, developers and testers to assess the stability and reliability of hardware and software under high load [43].

Providing over 270 different stress options, called stressors, in areas such as CPU load, memory allocation and access, and disk I/O, which can be used individually or in combination to create a realistic load scenario for a system [74]. In addition, stress-ng provides options to control the duration and intensity of the stressors [6].

Stress-ng is a command line program that can also be called from other programs, such as TestSuite, using the `fork` [51] and `execlp` [50] commands.

In order to generate the additional system load in the CPU, memory, I/O and timer areas during a test, stress-ng was used as it is a proven tool for generating stress with easy control. The stressors used are briefly introduced and described below. A more detailed description of all stressors and options of stress-ng can be found in its manual [6].

3.4.1.1 CPU Load

As per the requirements, the CPU load must be generated in both user space and kernel space for which different stressors have to be used. Additionally, section 3.4 specifies the need to create a load through real-time processes.

3.4.1.1.1 Generation of CPU Load in User Space

To generate CPU load in the user space of the system, the stressor 'cpu' from stress-ng is used. This stressor performs complex mathematical calculations, including integer and floating-point calculations, matrix operations, and checksum calculations, which place a heavy load on the CPU.

A worker of this stressor fully utilizes one CPU core. stress-ng provides an option for this particular stressor to limit the CPU load to an integer value between 0% and 100%. To fully utilize the system, stress-ng creates as many workers as the computer system has cores.

Figure 3.8 displays the 1-minute execution of 16 'cpu' stressors on the HPC1 system, which has 16 logical CPU cores. It is evident that the 'cpu' stressor fully utilizes this system. The load is generated in user space.

3.4.1.1.2 Generation of CPU Load in Kernel Space

The 'get' stressor is used to generate CPU load in kernel space. This calls various system calls, which leads to a predominant CPU load in kernel space. System calls such as `getpid` (retrieving the PID of the process [52]), `getuid` (retrieving the user ID of the process [54]) or `gettimeofday` (retrieving the current time [53]) are used.

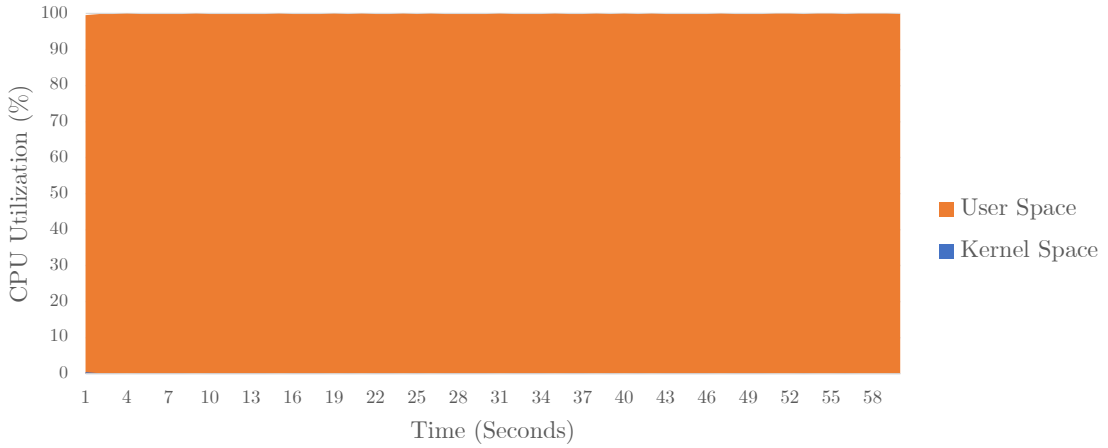

Figure 3.8: CPU Utilization during Execution of 16 stress-ng 'cpu' stressors on HPC1.

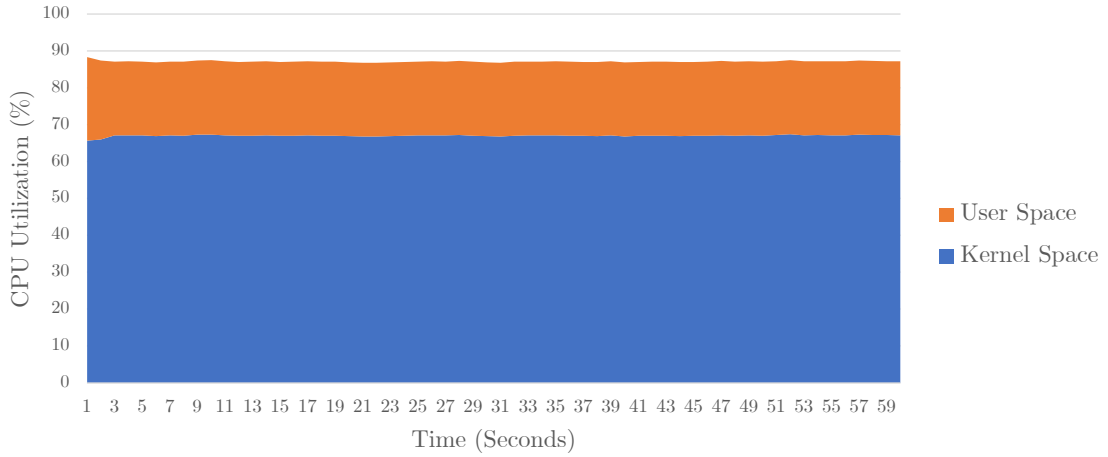

Figure 3.9: CPU Utilization during Execution of 16 stress-ng 'get' stressors on HPC1.

Figure 3.9 displays the 1-minute execution of 16 'get' stressors on the HPC1 system. The stressor utilizes approximately 87.5% of the total system load, with 20.1% generated in user space and 67.4% generated in kernel space.

As can be derived from Figure 3.9, this stressor does not fully utilize a CPU core. However, it still generates 60% to 70% of load in kernel space on a CPU core, making it suitable for generating CPU load in kernel space during tests.

3.4.1.1.3 Generation of CPU Load by Real-Time Processes

stress-ng allows for the specification of a scheduling policy and priority as additional options. This allows each stressor to be executed as a real-time process. For generating load, the CPU stressor described in section 3.4.1.1.1 was used with the `SCHED_FIFO` scheduling policy described in section 3.1.2.2.2 and a scheduling priority of 50. The scheduling priority was intentionally set lower than that of the TestSuite because communication processes have a higher priority in the distributed test system.

3.4.1.2 Memory Load

To generate memory load, the stress-ng 'bigheap' stressor is utilized. The stressor grows its heap by reallocating memory. If the Out Of Memory (OOM) killer on Linux, which is a process emplyed by the kernel when low memory is available on the system and kills one or more processes to resolve the situation [75], kills the process or the allocation fails then the stressor starts again.

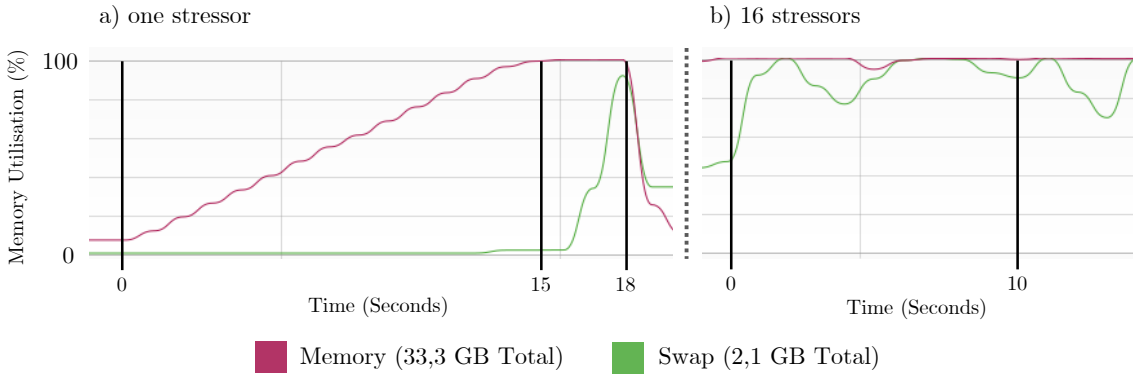

Figure 3.10: Memory Utilization during Execution of stress-ng 'bigheap' stressors on HPC1.

Figure 3.10 displays the memory utilization over time for a 'bigheap' stressor on HPC1 and compares it with the execution of 16 'bigheap' stressors that utilize all cores of the investigated computer system. The procedure for memory utilization is shown. The stressor requests more memory between 0 and 15 seconds until the memory is full, after which the system's swap increases. If the system has no more memory, the OOM killer terminates the process. The stressor is then restarted. If

16 'bigheap' stressors are executed simultaneously, this procedure overlaps.

The memory stressor, like any other process, also utilizes the system's CPU. Executing a memory stressor results in 100% CPU utilization for one core.

3.4.1.3 I/O Load

The 'hdd' stressor is used to generate I/O load. It performs continuous writes to the system's hard disk.

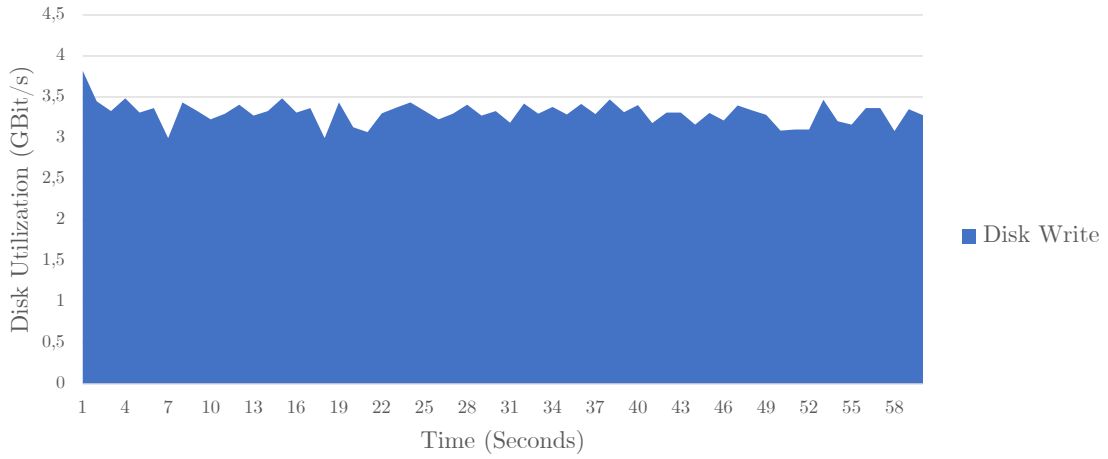

Figure 3.11: Disk Utilization during Execution of one stress-ng 'hdd' stressors on HPC1.

Figure 3.11 shows the hard disk load for a 1-minute execution of an hdd stressor. It can be seen that this performs write operations at an average data rate of around 3.2 GBit/s. Furthermore, the data rate can fluctuate between 3 GBit/s and 3.8 GBit/s. The reasons for this is that the stressor works internally with different methods and iterates through them. The methods differ, for example, in the size of the files that are written to the hard disk. The execution of an hdd stressor leads to 100% CPU utilization of a CPU core.

Linux has I/O scheduling, which reorders and groups requests based on the logical address of the data, with the goal of improving I/O throughput [81]. To maximize I/O device stress, I/O scheduling was disabled by selecting the "none" scheduling policy.

stress-ng does not provide a default way to limit I/O stress by limiting the data rate. To achieve this limit, Control Groups (cgroup) were used. Control groups are a kernel function that can limit the resource utilization of processes. In some tests in this work, control groups were used to limit the maximum data rate of the HDD stressor. Further information on the used cgroup2 can be found in its documentation [38].

3.4.1.4 Interrupt Load

The 'timer' stressor was utilized to generate load in the domain of interrupts. This produces timer events at a rate of 1 MHz, resulting in timer clock interrupts. Each timer event is then intercepted by a signal handler. The purpose of this stressor is to test the system under high interrupt load [74]. The execution of one timer stressor also fully utilizes a CPU core to 100%. Additionally, the system is loaded by the processing of the signal handler. The entire system's utilization was examined using an HPC as an example, which amounts to approximately 6.3%.

It is important to note that the timer results generated in this manner differ from interrupts generated by external hardware. Although both events are asynchronous [5], there is a significant difference between them. Timer events are generated periodically by the operating system, while hardware interrupts are triggered unpredictably by external devices such as network devices [70]. Signal handlers execute timer events in the context of the respective process, while interrupt handlers process hardware interrupts directly in the kernel [5].

3.4.2 iPerf2

iPerf2 is a tool for measuring bandwidth between two hosts on IP networks. It works according to the client-server principle, where the client is the sender and the server is the receiver. iPerf supports protocols such as TCP and UDP.

Various parameters can be specified when performing a measurement. These include the test duration, the datagram size or a target bandwidth. At the end of each

measurement, iPerf reports the achieved bandwidth, jitter and packet losses. More information about iPerf can be found in its documentation [36].

In this work, iPerf is used with the UDP protocol for a given target and datagram size. The only goal is to generate network traffic. To measure reliability and performance characteristics, only the TestSuite presented in Section 3.3 is used as part of the thesis.

NOCH NICHT IN DIESEM TEIL:

- Kapitel 4 – Tests zur Zuverlässigkeit mit Switch und iHawk Topologien
- Kapitel 5 – Performancemessungen unter versch. Betriebszuständen
- Kapitel 6 – Conclusion, Zusammenfassung der Konfigurationen / Punkte woduch UDP geeignet ist
- Kapitel 7 – Ausblick: Fragmentierung und DPDK (User Space Driver)
- Code im Anhang

4 Bibliography

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

[2] Arseny Kapoulkine. pugixml 1.14 Quick Start Guide, 2023. Accessed on 23.11.2023. URL: `https://pugixml.org/docs/quickstart.html`.

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

[4] Board Infinity. A Quick Guide to STAR Topology, 2023. Accessed on 02.12.2023. URL: `https://www.boardinfinity.com/blog/star-topology/`.

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

[6] Canonical Ltd. Ubuntu Manpage: stress-ng, 2023. Accessed on 13.10.2023. URL: `https://manpages.ubuntu.com/manpages/jammy/en/man1/stress-ng.1.html`.

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

[8] Cisco Systems, Inc. Cisco Business 350 Series Managed Switches Data Sheet, 2022. Accessed on 14.12.2023. URL: `https://www.cisco.com/c/en/us/products/collateral/switches/business-350-series-managed-switches/datasheet-c78-744156.html`.

[9] Concurrent Real-Time. RedHawk Linux: Real-Time Linux Development Environment, 2017. Brochure for RedHawk Linux, highlighting its features and applications in various industries.

[10] Concurrent Real-Time. iHawk, 2023. Accessed on 14.12.2023. URL: https://concurrent-rt.com/products/hardware/ihawk/.

[11] Concurrent Real-Time. RedHawk Linux RTOS, 2023. Accessed on 19.12.2023. URL: https://concurrent-rt.com/products/software/redhawk-linux/.

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

[13] Joe Damato. Monitoring and Tuning the Linux Networking Stack: Sending Data, 2017. Accessed on 05.12.2023. URL: https://blog.packagecloud.io/monitoring-tuning-linux-networking-stack-sending-data/.

[14] EKF Elektronik GmbH. *SN5-TOMBAK - CompactPCI Serial Dual-Port SFP+ 10Gbps Ethernet NIC*. EKF Elektronik GmbH, 2016. Document No. 8117.

[15] EKF Elektronik GmbH. *SC5-FESTIVAL - CompactPCI Serial CPU Card*. EKF Elektronik GmbH, 2023. Document No. 8459.

[16] 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.

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

[18] 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.

[19] 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.

[20] Behrouz A. Forouzan and Sophia Chung Fegan. *Data Communications and Networking*. McGraw-Hill, 4 edition, 2007.

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

[22] Elliotte Rusty Harold. *XML Bible.* Wiley, 2001.

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

[24] 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.

[25] 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.

[26] 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.

[27] Intel Corporation. Jumbo Frames and Jumbo Packets Notes, 2019. Accessed on 18.12.2023. URL: https://www.intel.com/content/www/us/en/support/articles/000006639/ethernet-products.html.

[28] Intel Corporation. Advanced Driver Settings for Intel Ethernet 10 Gigabit Server Adapters, 2020. Accessed on 13.11.2023. URL: https://www.intel.com/content/www/us/en/support/articles/000005783/ethernet-products.html.

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

[30] Intel Corporation. Intel Core i9-13900K Processor, 2023. Accessed on 09.11.2023. URL: https://ark.intel.com/content/www/us/en/ark/products/230496/intel-core-i9-13900k-processor-36m-cache-up-to-5-80-ghz.html.

[31] Intel Corporation. Intel Ethernet Converged Network Adapter X520-DA2, 2023. Accessed on 19.11.2023. URL: https://ark.intel.com/content/www/us/en/ark/products/39776/intel-ethernet-converged-network-adapter-x520-da2.html.

[32] Intel Corporation. Intel Ethernet Converged Network Adapter X540-T2, 2023. Accessed on 19.11.2023. URL: https://ark.intel.com/content/www/us/en/ark/products/58954/intel-ethernet-converged-network-adapter-x540-t2.html.

[33] Intel Corporation. Intel Ethernet Network Adapter X710-T2L, 2023. Accessed on 19.11.2023. URL: https://ark.intel.com/content/www/us/en/ark/products/189463/intel-ethernet-network-adapter-x710-t2l.html.

[34] Intel Corporation NEX Cloud Networking Group (NCNG). Intel ethernet 700 series linux performance tuning guide. Technical Report 334019, Intel Corporation, 2023. Document No.: 334019 Rev.: 1.1.

[35] 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.

[36] iPerf Development Team. iPerf Homepage, 2023. Accessed on 01.12.2023. URL: https://iperf.fr.

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

[38] Kernel Development Community. Control Group v2, 2023. Accessed on 14.10.2023. URL: https://docs.kernel.org/admin-guide/cgroup-v2.html.

[39] Kernel Development Community. Interface statistics - The Linux Kernel Documentation, 2023. Accessed on 23.11.2023. URL: https://docs.kernel.org/networking/statistics.html.

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

[41] Kernel Development Community. nmon for Linux, 2023. nmon Development Team. URL: https://nmon.sourceforge.io/pmwiki.php.

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

[43] Colin Ian King. stress-ng. A stress-testing Swiss army knife., 2019. Presentation. Accessed on 13.10.2023. URL: https://elinux.org/images/5/5c/Lyon-stress-ng-presentation-oct-2019.pdf.

[44] James F. Kurose and Keith W. Ross. *Computer Networking: A Top-Down Approach*. Pearson, 8 edition, 2021.

[45] Christoph Lameter. Numa (non-uniform memory access): An overview: Numa becomes more common because memory controllers get close to execution units on microprocessors. *Queue*, 11(7):40–51, jul 2013. doi:10.1145/2508834.2513149.

[46] Lenovo (Beijing) Limited. ThinkSystem Marvell QL41132 and QL41134 10GBASE-T Ethernet Adapters, 2021. Accessed on 19.11.2023. URL: https://lenovopress.lenovo.com/lp0902-marvell-ql41132-ql41134-10gbase-t-ethernet-adapters.

[47] Yadong Li, Linden Cornett, Manasi Deval, Anil Vasudevan, and Parthasarathy Sarangam. Adaptive interrupt moderation, 2015.

[48] Linux Manual Page Contributors. chrt(1) — Linux Manual Page. Man7.org, 2023. Accessed on 18.12.2023. URL: https://man7.org/linux/man-pages/man1/chrt.1.html.

[49] Linux Manual Page Contributors. clock_gettime(3) — Linux Manual Page. Man7.org, 2023. Accessed on 05.12.2023. URL: https://man7.org/linux/man-pages/man3/clock_gettime.3.html.

[50] Linux Manual Page Contributors. exec(3) — Linux Manual Page. Man7.org, 2023. Accessed on 13.10.2023. URL: https://man7.org/linux/man-pages/man3/exec.3.html.

[51] Linux Manual Page Contributors. fork(2) — Linux Manual Page. Man7.org, 2023. Accessed on 13.10.2023. URL: https://man7.org/linux/man-pages/man2/fork.2.html.

[52] Linux Manual Page Contributors. getpid(2) — Linux Manual Page. Man7.org, 2023. Accessed on 13.10.2023. URL: https://man7.org/linux/man-pages/man2/getpid.2.html.

[53] Linux Manual Page Contributors. gettimeofday(2) — Linux Manual Page. Man7.org, 2023. Accessed on 13.10.2023. URL: https://man7.org/linux/man-pages/man2/gettimeofday.2.html.

[54] Linux Manual Page Contributors. getuid(2) — Linux Manual Page. Man7.org, 2023. Accessed on 13.10.2023. URL: https://man7.org/linux/man-pages/man2/getuid.2.html.

[55] 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.

[56] 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.

[57] Linux Manual Page Contributors. send(2) — Linux Manual Page. Man7.org, 2023. Accessed on 05.12.2023. URL: https://man7.org/linux/man-pages/man2/send.2.html.

[58] 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.

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

[60] 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/.

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

[62] Microsoft Corporation. Interrupt Moderation, 2021. Accessed on 13.11.2023. URL: https://learn.microsoft.com/en-us/windows-hardware/drivers/network/interrupt-moderation.

[63] 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.

[64] National Aeronautics and Space Administration. Endeavour Configuration Details, 2021. Accessed on 14.12.2023. URL: https://www.nas.nasa.gov/hecc/support/kb/endeavour-configuration-details_662.html.

[65] National Aeronautics and Space Administration. Skylake Processors, 2021. Accessed on 14.12.2023. URL: https://www.nas.nasa.gov/hecc/support/kb/skylake-processors_550.html.

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

[67] PICMG. CompactPCI Serial Overview, 2023. Accessed on 10.12.2023. URL: https://www.picmg.org/openstandards/compactpci-serial/.

[68] Pawan Prakash, Myungjin Lee, Y. Charlie Hu, Ramana Rao Kompella, and Twitter Inc. Jumbo Frames or Not: That is the Question! Technical Report 13-006, Purdue University Purdue University, 2013.

[69] 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.

[70] Red Hat, Inc. Hardware interrupts - Red Hat Enterprise Linux for Real Time 8, 2023. Accessed on 13.10.2023. URL: https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux_for_real_time/8/html/reference_guide/chap-hardware_interrupts.

[71] 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.

[72] Red Hat, Inc. Priorities and Policies - Red Hat Enterprise Linux for Real Time 7, 2023. Accessed on 18.12.2023. URL: https://access.redhat.com/documentation/de-de/red_hat_enterprise_linux_for_real_time/7/html/reference_guide/chap-priorities_and_policies.

[73] 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.

[74] Red Hat, Inc. Stress testing real-time systems with stress-ng - Red Hat Enterprise Linux for Real Time 8, 2023. Accessed on 13.10.2023. URL: https://access.redhat.com/documentation/de-de/red_hat_enterprise_linux_for_real_time/8/html/optimizing_rhel_8_for_real_time_for_low_latency_operation/assembly_stress-testing-real-time-systems-with-stress-ng_optimizing-rhel8-for-real-time-for-low-latency-operation.

[75] José Rocha. Linux Out of Memory killer, 2023. Accessed on 13.10.2023. URL: https://neo4j.com/developer/kb/linux-out-of-memory-killer/.

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

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

[78] Super Micro Computer, Inc. *X11DPi-N X11DPi-NT User's Manual*. Super Micro Computer, Inc., 2021. Document No. 8459.

[79] Andrew S. Tanenbaum and David J. Wetherall. *Computer Networks*. Prentice

Hall Press, USA, 5th edition, 2010.

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

[81] Ubuntu Wiki. IOSchedulers, 2023. Accessed on 14.10.2023. URL: https://wiki.ubuntu.com/Kernel/Reference/IOSchedulers.

[82] UserBenchmark. Comparison of Intel Core i3-7100 and Intel Core i7-3770S, 2023. Accessed on 10.12.2023. URL: https://cpu.userbenchmark.com/Compare/Intel-Core-i3-7100-vs-Intel-Core-i7-3770S/3891vsm2218.

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

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

[85] 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