TU WIEN Informatics

Design and Limitations of a Software-based TSN End Station

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Technische Informatik

eingereicht von

Christoph Lehr, BSc
Matrikelnummer 01525189

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Prof.Dr. Wolfgang Kastner
Mitwirkung: Dipl-Ing. Patrick Denzler, MSc
Dipl-Ing. Dr. Thomas Frühwirth

Wien, 18. Jänner 2023

Christoph Lehr | Wolfgang Kastner

Technische Universität Wien
A-1040 Wien ▪ Karlsplatz 13 ▪ Tel. +43-1-58801-0 ▪ www.tuwien.at

TU WIEN Informatics

Design and Limitations of a Software-based TSN End Station

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Computer Engineering

by

Christoph Lehr, BSc
Registration Number 01525189

to the Faculty of Informatics

at the TU Wien

Advisor: Prof.Dr. Wolfgang Kastner
Assistance: Dipl-Ing. Patrick Denzler, MSc
Dipl-Ing. Dr. Thomas Frühwirth

Vienna, 18th January, 2023

Christoph Lehr	Wolfgang Kastner

Technische Universität Wien
A-1040 Wien ▪ Karlsplatz 13 ▪ Tel. +43-1-58801-0 ▪ www.tuwien.at

Erklärung zur Verfassung der Arbeit

Christoph Lehr, BSc

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 18. Jänner 2023

Christoph Lehr

Danksagung

Ich möchte mich hiermit herzlich bei Professor Kastner bedanken, welcher mir nach meiner Bachelorarbeit die Möglichkeit gab, meine Diplomarbeit mit Unterstützung der von ihm geleiteten Arbeitsgruppe zu verfassen. Ich möchte mich auch dafür bedanken, dass mir diese Arbeit die Möglichkeit gab, mein Wissen für die genutzten Technologien zu erweitern und zu vertiefen.

Weiteres möchte ich mich herzlich bei Patrick Denzler und Thomas Frühwirth bedanken, die mir immer mit Rat und Tat zur Seite standen, um diese Arbeit voranschreiten zu lassen.

Des weiteren möchte ich mich bei Martin Schöberl von der Technischen Universität Dänemark (DTU) und Emad Maroun für die Unterstützung bei verschiedenen Problemen mit T-CREST Patmos bedanken.

Ich möchte auch Anna Knabe meinen herzlichsten Dank aussprechen. Sie hat während der Erstellung dieses Dokuments unzählige Male Korrektur gelesen und lektoriert.

Schlussendlich möchte ich mich noch bei meiner Familie, meinen Freunden und Kommilitonen bedanken, welche mir immer wieder ein offenes Ohr geschenkt haben und mich in meinen Bestrebungen unterstützt haben.

Kurzfassung

Das Industrial Internet of Things (IIoT) ist eine neue Entwicklung, die immer mehr Geräte miteinander vernetzt. Es konzentriert sich auf die Erfassung von Daten in industriellen Prozessen und Verarbeitungsanlagen. In den genannten Bereichen gehören Feldbusse, wie das Controller Area Network (CAN) und Profibus, zu den am weitesten verbreiteten Netzwerktechnologien. Ihnen fehlt leider die Bandbreite für die Datenerfassung, die für das IIoT und Industrie 4.0 benötigt wird. Um diesen Anforderungen gerecht zu werden, werden immer mehr Ethernet-basierte Protokolle verwendet. Eine Lösung ist beispielsweise Time-Sensitive Networking (TSN). Ethernet ist das Rückgrat der modernen Informationstechnologie (IT) und ein weit verbreiteter Standard. Was diese Technologie für das IIoT interessant macht, ist, dass moderne Ethernet-Geräte im Gegensatz zu Feldbussen, wie beispielsweise CAN, signifikant höhere Übertragungsgeschwindigkeiten ermöglichen. Des weiteren sind Ethernet-Komponenten vergleichsweise kostengünstig durch ihrer Verbreitung in der IT.

Diese Arbeit befasst sich mit verschiedenen Herausforderungen, die bei der Implementierung einer TSN-Endstation auftreten. Zunächst wurde ein Taktsynchronisationsalgorithmus implementiert, der es der Endstation erlaubt, sich mit dem TSN-Netzwerk zu synchronisieren. TSN organisiert dann die Ethernet-Pakete entsprechend ihrer Priorität in verschiedenen Warteschlangen. Daher wurde ein echtzeitfähiges Speicherverwaltungssystem implementiert. Des Weiteren wurde eine Netzwerkschnittstelle entwickelt, um Nachrichten präzise zu vordefinierten Zeitpunkten zu übertragen. Alle diese Aspekte wurden einzeln bewertet und dann in einem echtzeitfähigen TSN-Netzwerkstack kombiniert. Wir haben das System für die T-CREST Patmos-Plattform entwickelt, die eine Ausführung in Echtzeit ermöglicht. Sie ermöglicht außerdem eine Worst-Case Execution Time (WCET) Analyse für maximiale Laufzeit des von uns vorgeschlagenen Systems.

Abstract

The Industrial Internet of Things (IIoT) is an emerging development, connecting an ever increasing number of devices. It focuses on the collection of data in industrial processes and processing facilities. In these areas, field buses like Controller Area Network (CAN) and Profibus, belong to the most common networks, but they lack the bandwidth required for broader data collection. To satisfy this demand, more and more Ethernet-based protocols are used. Ethernet is the backbone of modern Information Technology (IT) infrastructure and a widely used standard. What makes this technology interesting for the IIoT is that modern Ethernet devices have transport speeds of multiple Gigabits per second (Gbit/s). That is several orders of magnitude higher than CAN, where speeds reach a maximum bandwidth of around one Megabit per second (Mbit/s). The process of integrating Ethernet-based technologies from IT into the production facilities, which are also called Operational Technology (OT), is typically referred to as the IT/OT convergence. As standard Ethernet cannot provide any real-time guarantees, Time-Sensitive Networking (TSN) was developed. However, only a very limited number of devices currently exist, particularly regarding TSN end stations.

This thesis addresses several challenges that arise when implementing a TSN end station. First, a clock synchronization algorithm had to be implemented, allowing the end station to synchronize with the TSN network. Second, TSN organizes Ethernet packets in different queues according to their priority. Therefore, a real-time memory management system was implemented. Third, a network interface was developed to transmit messages precisely at pre-defined points in time. All these aspects were individually evaluated and then combined in a real-time capable TSN network stack. We created the system for the T-CREST Patmos platform, which allows time-predictable execution. This platform allows the Worst-Case Execution Time (WCET) analysis of our proposed system.

Contents

CHAPTER 1

Introduction

As the Internet of Things (IoT) continues to evolve, the number of interconnected devices grow along in an unprecedented way. This development is not limited to households and offices, but also takes place in industrial automation. IoT in this context is often referred to as the Industrial Internet of Things (IIoT) which focuses on the collection of data in industrial processes and processing facilities. Traditionally field buses like Controller Area Network (CAN) and Profibus [1] are common network protocols in the industry, however, they lack bandwidth for broader data collection. Therefore, recent developments focused on introducing more bandwidth in industrial networks while preserving the high reliability of traditional field buses. The most promising solution is Time-Sensitive Networking (TSN), which is based on the Ethernet protocol [2]. To satisfy this demand, more and more Ethernet-based protocols are used, one solution is TSN.

Ethernet is the backbone of modern Information Technology (IT) infrastructure and a widely used standard. Reasons for its broad usage are the relatively low prices for equipment and that it is easy to expand existing networks. What makes this technology interesting for the IIoT is that modern Ethernet devices have transport speeds of multiple Gigabits per second (Gbit/s). That is several orders of magnitude higher than CAN, where speeds reach a maximum bandwidth of around one Megabit per second (Mbit/s). The process of integrating Ethernet-based technologies from Information Technology (IT) into the production facilities, which are also-called the Operational Technology (OT), is typically referred to as the IT/OT convergence [3][4].

The IT/OT convergence describes the transformation of the automation pyramid into the automation pillar [3][5], which is depicted in Figure 1.1. On the left side is the automation pyramid, which consists of multiple layers. It spans from the *Field Level* containing sensors and actuators, over the *Control Level* mainly built from Programmable Logic Controllers (PLCs), the *Supervisory Level* providing Supervisory Control and Data Acquisition (SCADA) applications, the *Planning Level* for the Manufacturing Execution System (MES), to the *Management Level* with its Enterprise Resource Planning (ERP)

software. On the right side the so-called automation pillar is depicted, which shows the goal of this transition: separating industrial automation into a distributed automation layer and an automation cloud connected by the high speed-low latency network TSN. A reason for this transition is to ease the interaction between the layers as IT and OT start operating on the same network and reduce the complexity.

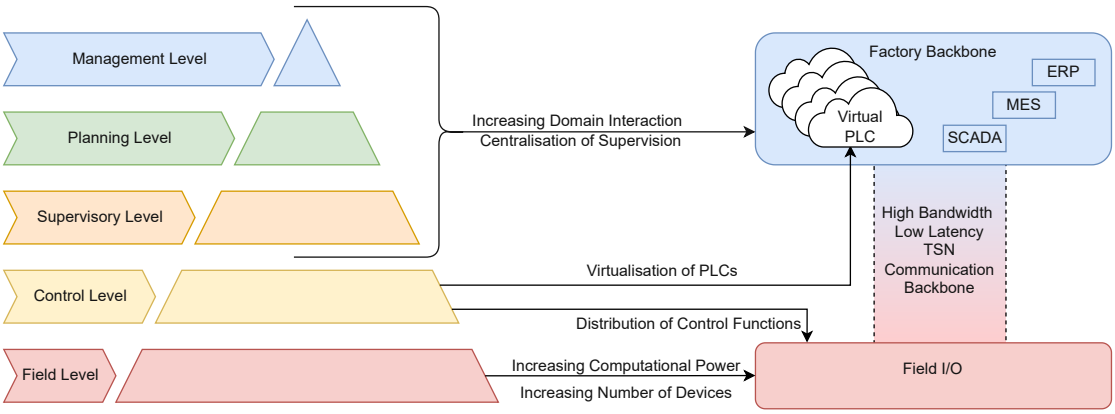


Figure 1.1: Transition from automation pyramid to automation pillar, adapted from [5]

TSN is an open protocol that builds upon standard Ethernet to allow easy integration with Commercial of the Shelf (COTS) IT systems. This is in direct contrast to most Ethernet-based protocols used in industrial automation, like ProfiNET [6] or Ethernet Powerlink [7], which require dedicated hardware to participate in the network. Like other protocols used in industrial automation, TSN provides transmission capabilities for Real-Time (RT) and non-RT traffic. TSN is standardised by the Institute of Electrical and Electronics Engineers (IEEE), which provides the benefit that no single vendor has sole control over the specification. This prevents a vendor lock-in where only one or a small group of suppliers provide certified devices. Nevertheless, having a proprietary protocol is not a drawback. If specifications are only controlled by a single company, the standard is often quite stable. Nonetheless, the state of the art changes and new requirements will manifest, therefore, specifications need to be adapted and added. As TSN is published by IEEE, manufacturers and technical partners can steer the standard development and improve the documents to be more relevant for future uses.

Figure 1.2 depicts an example network utilising TSN so end stations can communicate with each other and the management clients. The network allows transmitting RT and non-RT messages over the same infrastructure. In the case of TSN, a network-wide schedule [8] defines when each member is allowed to transmit data. As frames with mixed criticality use the same communication medium, the single members need to synchronise with each other. Therefore, each member has to synchronise its clock to achieve consensus on who can access the medium. To communicate with each other, every member requires certain network functionalities, which the network stack manages and provides to the various applications running on an end stations. Central tasks of it are serialising and deserialising protocol data and distributing the received data to the

applications. The network stack interacts with the driver of the TSN network interface to access the transmission medium. The driver configures the interface and manages the interface's memory to store incoming and outgoing messages. Further, it assigns the messages to their queues.

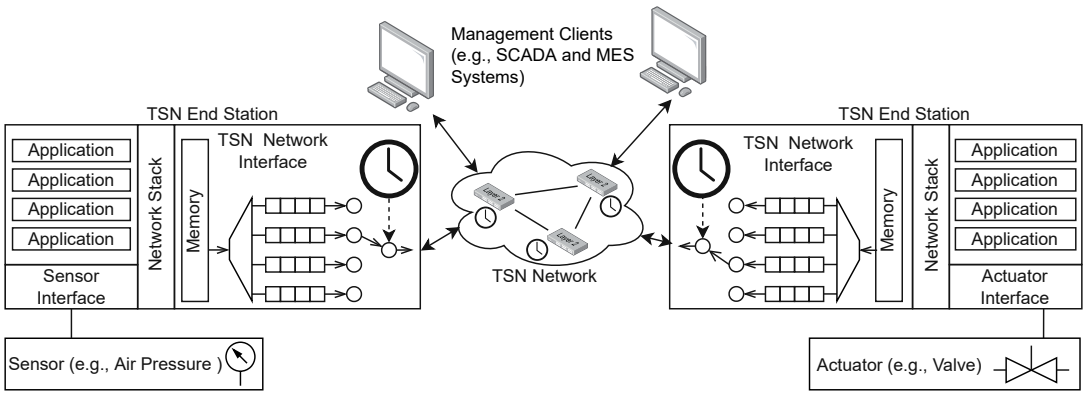


Figure 1.2: Example industrial application setup using TSN communication

1.1 Problem Statement and Aim of the Work

As TSN is still a relatively new protocol, building demonstration networks helps to evaluate the current state of the art and investigate different approaches. Most of these networks focus on the network hardware like switches [9] or creating network schedules [10][11], but seldom on the connected end stations. As TSN end stations are essential to provide deterministic transmission, they can also be the source for blocking the network with best-effort traffic if TSN is not implemented correctly.

Implementing TSN on an end station can be done in hardware or software. However, hardware solutions are more common nowadays [9], and software implementations have not yet received much attention from industry or academia [12]. A benefit of a software implementation is its flexibility on different platforms and the possibility of adjustments if required. Within this thesis, the focus lies on software-based TSN end stations and their implementation and configuration.

Therefore, the aim of this thesis is to identify time-critical elements of a software-based TSN end-system and investigate a suitable network-stack architecture and its limitations. A specific interest lies in the evaluation of the Worst-Case Execution Time (WCET) behaviour the time-critical elements of the proposed network stack architecture.

The thesis shall answer the following research questions:

RQ1: How shall a network stack be structured to support real-time and non-real-time traffic?

RQ2: What accuracy can be expected from a clock synchronisation algorithm partly implemented in software?

RQ3: What buffer management algorithm provides the optimal trade-off between time complexity and memory overhead?

RQ4: What limitations occur when realising a software-based TSN network interface?

1.2 Delimitations

There are further building blocks for the implementation of a TSN end station that are not part of this thesis. One of them is the implementation of Stream Reservation Protocol (SRP), which can be utilized to dynamically reserve network resources for streams. SRP can be used as the basis for dynamically allocating time slots or reconfiguring traffic shaping algorithms to guarantee lower latency or better throughput for certain streams. Another topic is Frame Replication and Elimination for Reliability (FRER) [13], which is used to improve reliable transport by utilizing redundant paths in the network. Maryam Pahlevan and Roman Obermaisser [14] further investigated the usage of FRER. Elements like Path Control and Reservation (PCR) [15], Cyclic queuing and forwarding (CQF) [16] and Per-Stream Filtering and Policing (PSFP) [17] are not included, as these elements are related to the network hardware like routers and switches and not to end stations. Additionally, device configuration via Network Configuration (NETCONF) and implementation of a YANG model are out of scope.

1.3 Contribution

Beyond answering the above research questions, this thesis provides several concrete scientific and engineering contributions in the context of TSN end stations. They are grouped by topic and briefly summarized in the following paragraphs.

Contributions in the area of real-time network stacks primarily have an engineering character. They include the design of a network stack supporting RT and non-RT traffic, a comprehensive WCET analysis of all relevant functions, and the corresponding evaluation that verifies that the theoretical upper bounds are valid.

Regarding time synchronization, the thesis provides a generalized Precision Time Protocol (gPTP) implementation in the form of a time synchronization service for the T-CREST platform. It enables clock accuracy of about 40 µs. Furthermore, it supports the Best Master Clock Algorithm (BMCA).

Several of our contributions address buffer management in RT systems. We extend existing classifications of memory management approaches as well as summarize and – where lacking – deduce the limiting behaviour regarding computation time and memory overhead of dynamic storage allocation algorithms in big O notation. Furthermore, we optimize the data structures used by the Two-Level Segregated Fit (TLSF) algorithm and provide the corresponding WCET analysis and evaluation.

Finally, we provide an implementation of a TSN driver for the T-CREST platform that primarily focuses on the traffic scheduling mechanisms defined in IEEE 802.1Q. The implementation includes a discussion of space-optimized data structures and is WCET analysed and evaluated.

During the creation of this thesis, several code constructs were designed and implemented. All code constructs, the programs used for evaluation and the implementation can be accessed via git [18]. This includes a fully functional RT network stack and a service for time synchronisation. In addition to a standard Ethernet interface, it provides a functional TSN network interface. Both interfaces can be configured to utilise a dynamic memory management system.

In addition, this repository holds a hardware configuration for the used evaluation platform T-CREST [19] (Patmos [20]). This configuration adds a second network interface. Further, it adds a dedicated timer that is required for correct operation of the TSN interface.

1.4 Structure of the Work

The remaining thesis is structured in the following chapters:

- **Chapter 2: State of the Art**
  In this chapter, we give a brief introduction to the technical background required for this thesis. This covers Real-Time Systems (RTS), WCET analysis, T-CREST Patmos and TSN. Further, we provide an overview of scientific work related to this thesis.

- **Chapter 3: Methodological Approach**
  In this chapter, we provide the methodological approach we have applied.

- **Chapter 4: Time Predictable Network Stack Design**
  In this chapter, we focus on the principals of designing and implementing a network stack. This covers the various layers of the networking models and protocols belonging to them. Further, we investigate how such a network stack can be made WCET analysable and what is required for RT operation. In addition, we present our design, evaluate it, and discuss our findings.

- **Chapter 5: Time Synchronisation**
  This chapter focuses on the realisation of time synchronisation in a deterministic

network. We take a look at the definition of gPTP as this is used for TSN. Furthermore, we provide a design for a time synchronisation service, which synchronises the end station and provides accurate time readings to the application. Finally, we evaluate the design and discuss the results.

- **Chapter 6: Buffer Management**
  A fundamental task when working with network interfaces is to provide temporary buffers to store data for incoming and outgoing messages. Therefore, this chapter focuses how such a mechanism can be realised for an RTS. We introduce a classification of such systems and algorithms used for Dynamic Storage Allocation (DSA). Further, we design and implement a buffer management system for our proposed time predictable network stack. We evaluate this system using static WCET tooling and runtime analysis.

- **Chapter 7: Time-Sensitive Networking**
  In this chapter, we provide a deeper introduction to TSN. Derived from the specification we design our TSN interface and implement a driver for it. We test the interface and analyse it using a static WCET calculation methods. Finally, we present the gathered results.

- **Chapter 8: Conclusion**
  In this chapter, we conclude the findings of the previous chapters and discuss them in the context of the above research questions.

CHAPTER 2

State of the Art

In this chapter, we provide the initial technical background relevant for this thesis. This includes the introduction of essential terms which the individual chapter build upon. We introduce the core concepts of RTSs, in Section 2.1. A fundamental tool for designing RTSs is the timing analysis, the basics of it get established in Section 2.2. Further, we introduce T-CREST and Patmos in Section 2.3, which is the used processor to realise this thesis. As stated in Chapter 1, this thesis focuses on the design of a deterministic TSN end station and initial introduction to TSN is given in Section 2.4. In Section 2.5, we provide related work around the usage of TSN in various environments and technologies supporting TSN and deterministic RTSs.

Since the single topics discussed in this thesis build up on each other, but are not intertwined, we decided to introduce additional information when required. Therefore, the individual chapters provide further details on the technical background and related work of the topics.

2.1 Real-Time Systems

According to the definitions provided by Kopetz [21], a RTS is a computer system that changes its behaviour over time based on the stimuli from the environment. The change of the behaviour may depend on a physical process, e.g., the RTS may control a chemical reaction. Such processes dictates that the RTS has to act within certain intervals, also referred to as *deadlines*, in order to keep the technical process under control. If a required result has meaning after a deadline has passed, then this is a *soft deadline*. If the result has no meaning after a deadline has passed, then its called a *firm deadline*, and if severe consequences result from a missed deadline then this is called a *hard deadline*. If a system contains at least one hard deadline, it is called a *hard RTS* or *safety critical system*. Otherwise, such a system is typically referred to as a *soft RTS*. It should be noted that the design procedures for hard and soft RTSs are fundamentally different.

2.2 Timing Analysis and Worst-Case Execution Time

Typically, RTSs have stringent timing requirements, which are commonly dependent on the technical processes that is being controlled by them. Determining the upper bound of the execution time of control algorithms and other tasks is not trivial as many factors need to be considered and maybe restricted. The program or task that shall be analysed needs to be analysable at first, therefore, it is required that the program always terminates. In order for hard RTSs to be able to adhere to this, only restricted forms of programming are allowed, e.g., endless loops are forbidden.

In Figure 2.1, a possible runtime distribution is depicted. On the horizontal axis the application runtime is depicted and on the vertical axis how often said runtime occurs. The lowest observable execution time is the Best-Case Execution Time (BCET) and the highest observable execution time is the WCET. The highest possible upper bound is the WCET bound and is a theoretical value for the longest possible execution time. The WCET boundary largely depends on the worst case input for an application or task.

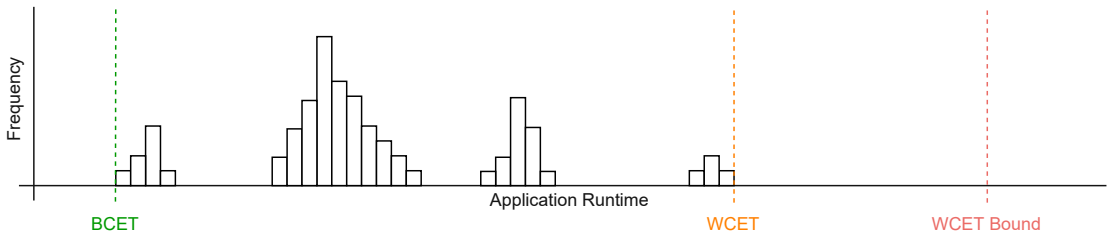


Figure 2.1: Application runtime distribution

These timing values can be determined by various approaches, Wilhelm et al. [22] summarised used techniques as well as tools for timing analysis. Commonly, measurement-based analysis is employed, but in general, this over-estimates the BCET and under-estimates the WCET. Unfortunately, these approaches are not appropriate for hard RTSs and therefore, for such systems static WCET analysis methods are employed. These techniques analyse the program flow as well as the possible behaviour and state of the hardware to provide a WCET bound by either inspecting the source code, machine code or an intermediate output format from the used compiler. In addition, the precision of the calculated upper bound is also dependent on the knowledge of the worst case input size, which may be hard to derive.

2.3 T-CREST and Patmos

Since embedded systems influence more and more areas of the industry and get used in safety-critical applications, new technologies were required to guarantee dependability. Such systems get used in aerospace and space programs as well as in the automotive and manufacturing industries. Therefore, universities across Europe worked together and founded the T-CREST [19] project to design a new hardware architecture for such systems.

This group has designed the Patmos processing core [20] to build a time-predictable multicore processor. It is Reduced Instruction Set Computer (RISC) based, which means that it utilises simpler instructions which can be executed faster. The RISC-V Instruction Set Architecture (ISA) is based on this philosophy as well as Central Processing Unit (CPU) cores designed by ARM®. In contrast, a Complex Instruction Set Computer (CISC) uses more complex instructions that execute operations at once, like loading a word from memory, adding add another word to it and storing it again in memory. Intel uses CISC as the basis of their x86 CPU architecture.

The T-CREST project ported the llvm compiler, commonly referred to as the Patmos compiler [23], for the used ISA. In Figure 2.2 the workflow of the build toolchain is depicted. The system uses C application code as well as user and system libraries as an input. These are compiled and linked to create an *elf* file that the Patmos CPU can execute. In addition, the compiler toolchain can create a *pml* file, which contains additional metadata. This file is used by *platin* for execution time analysis. Amongst the visualisation and flow facts that can be used for further analysis, the tool can export static WCET analysis data.

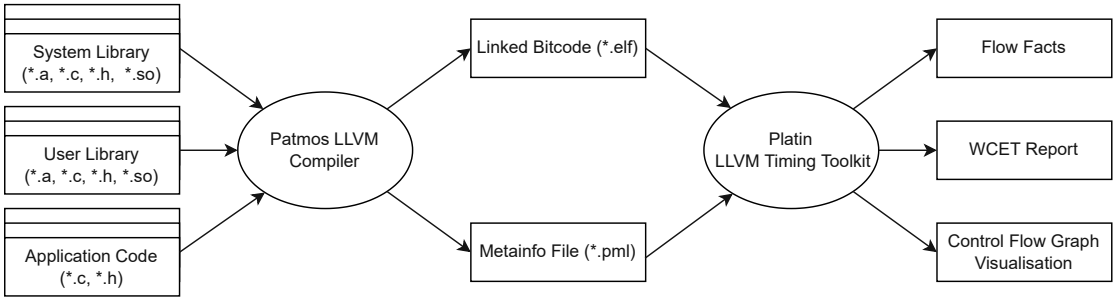


Figure 2.2: Patmos WCET tool-workflow

2.4 Time-Sensitive Networking

Time-Sensitive Networking (TSN) is a communication technology that builds upon Ethernet, which is the de-facto standard for wired communication. In contrast to other technologies building on Ethernet [2], like Time-Triggered Ethernet (TTEthernet) [24] or ProfiNET [6], TSN is compatible with standard Ethernet and can be used together with COTS components found in an IT department. Therefore, leading manufacturers and researchers see it as the technology to bridge the IT and OT, often referred to as the IT/OT convergence [4].

One of TSN's goals is to reduce latency and jitter between end stations. Therefore, TSN employs traffic scheduling techniques like rate limitations and time triggered-traffic [8]. This is part of the Media Access Control (MAC), which takes care of accessing a physical medium. The MAC of Ethernet implements Carrier-Sense Multiple Access with Collision Detection (CSMA/CD) [2] to control the access. As CSMA/CD is only able to detect

collisions but not resolve or avoid them, no real-time guarantees can be provided. In other words, standard Ethernet only provides Best Effort (BE) traffic guarantees. To overcome this deficiency, TSN was developed, which employs Time-Division Multiple Access (TDMA) instead. Therefore, a network schedule is required, which defines when each network device is allowed to send data. Protocols like TTEthernet [24] use so-called Virtual Links (VLs), which define precisely send windows for each application. TSN uses a simpler concept by assigning each frame to one of eight traffic classes, and the schedule defines which traffic classes are allowed to send.

For schedules to work as intended, time synchronisation is employed to establish a time base across a network. Therefore, a specialisation of Precision Time Protocol (PTP) [25] defined in IEEE 802.1AS [26] called gPTP is used to establish a global notation of time in a network. PTP is a decentralised protocol, which provides accuracy in the sub-microsecond range. Each network has a master clock, called the *grandmaster*, and all the clocks in the network synchronise with it.

2.5 Related Work

A small set of companies has already implemented TSN Hardware [27][1,2,3]. They provide the basic functionalities required to build a TSN network. TSN Intellectual Property Cores (IP-Cores) based on Intel Field-Programmable Gate Arrays (FPGAs) are also available[4]. To provide means for rapid and flexible customisation, Wei Quanet et al. have implemented *OpenTSN* [9]. This project provides an open-source IP-Core for a Network Interface Card (NIC) and a switch interface. The authors used Verilog to realise the design for the Xilinx Zynq FPGA line. In addition to the interface design, it provides a simple management protocol to configure the TSN interfaces.

One of the backbones of the transition from the automation pyramid to the automation pillar [3] is the usage of TSN [5]. Smart manufacturing is one of the key technologies of Industry 4.0, which requires interconnected devices of the IT and OT world. An existing open-source standard for safety-related applications is *openSAFETY*. This standard can be integrated on top of any network with deterministic behaviour. Gent et al. [28] showed that this is also possible utilising TSN.

An essential part of smart manufacturing is to adapt production facilities dynamically. This dynamic reconfiguration can optimise manufacturing steps or completely reconfigure the items produced. An aspect that shall be addressed is the safety of the system.

[1] https://www.tttech-industrial.com/wp-content/uploads/TTTech-Industrial_MFN-100.pdf, last accessed 6.12.2022

[2] https://www.br-automation.com/en/products/networks-and-fieldbus-modules/opc-ua-over-tsn/switch/0acst0521/, last accessed 6.12.2022

[3] https://hirschmann.com/de/Hirschmann/Hirschmann-News/Hirschmann_Managed_BOBCAT_Switches/index.phtml, last accessed 6.12.2022

[4] https://www.intel.com/content/www/us/en/industrial-automation/programmable/applications/automation/tsn.html, last accessed 6.12.2022

Therefore, Etz et al. [29] have defined a design guideline for engineering self-organising safety systems. The essential thought behind this system is to dynamically detect devices joining a network and derive a configuration for the network which allows safe operation. A human operator shall check the dynamically generated configuration to guarantee the correctness of said configuration. To prove the system's viability, the authors used TSN as the communication backbone as it provides RT guarantees. Again a vital element of this work is to build upon a vendor-neutral platform.

As already established, the IIoT and the IT/OT convergence lead to drastic changes in industrial automation. A newly established technology in this area is Open Platform Communication Unified Architecture (OPC UA), which is a service-oriented architecture for industrial automation[5]. One of the stepping stones to fully establishing OPC UA in this field is to provide end-to-end real-time machine-to-machine communication. Denzler et al. [30][31] investigated the WCET for network transmissions and the required adjustments for industrial applications. Both publications use TSN as the underlying communication network to guarantee deterministic traffic.

OPC UA is not the only used communication middleware, e.g., Data Distribution Service (DDS), Message Queue Telemetry Transport (MQTT), and Robot Operating System (ROS) are also widely adopted. These communication middlewares have various application fields. For instance, ROS is commonly used in robotics applications, whereas MQTT is typically found in building automation and IoT environments. Profanter et al. [32] provide an evaluation how these technologies compare.

Stricter control is required when switching from standard Ethernet to an RT-capable Ethernet protocol. Therefore, Wang et al. looked into defining a model for TSN-based Ethernet interfaces [12]. In the case of TSN, the interface driver needs to be able to obey a schedule. Since TSN uses traffic classes and transmission gates, it is not guaranteed that a frame will be transmitted instantly even if the communication channel is free. Therefore, the application layer and the driver need to work together. Frames ready to send shall not be modified further to guarantee correct transmission. Therefore, the application layer, specifically the application, has to lock the data to signal the driver what can be transmitted.

Another direction for control of the network is Software-Defined Networking (SDN). SDN is an approach to configure networks in a more dynamic fashion. The goal is to centralise processes and separate the processes of forwarding network packets and routing them. Silva et al. [33] investigated how SDN and TSN compare. Further, the authors analysed how these technologies satisfy the requirements imposed by Industry 4.0. Said et al. [34] further investigated this topic and propose how SDN can be utilised to configure and manage TSN networks.

In addition to the schedules, a central element of TSN is time synchronisation. Therefore, Kyriakakis et al. [35] designed a hardware timestamping unit for the Patmos ecosystem.

[5] `https://opcfoundation.org/about/opc-technologies/opc-ua/`, accessed 6.12.2022

The engine deserialises the Ethernet frames received and transmitted via the RX and TX channels of the Media Independent Interface (MII) connected to a Physical-Layer (PHY) chip. When the hardware detects a PTP message, a timestamp is generated that can be used to synchronise the clocks. The authors managed that two FPGAs with Patmos synchronise with an offset not exceeding 200 ns.

TSN is not the only RT Ethernet protocol, others are ProfiNET and EtherCAT. TTEthernet is another protocol for RT communication based on Ethernet. Therefore, Frühwirth et al. [36] designed a software-based end station for TTEthernet. The design is based on the AUTOSAR classic platform. The authors present an Ethernet stack to transmit mixed-criticality traffic over the same interface and network. Since TTEthernet is not only used in the automotive industry, Kyriakakis et al. [37] presented a time-predictable TTEthernet end-system for Patmos. The authors provide an open-source TTEthernet stack that can be WCET analysed.

CHAPTER 3

Methodological Approach

In this chapter, we introduce the used methodological approach within this thesis (cf. Section 3.1). In Section 3.2, we show how we applied our selected methodological approach. Finally, Section 3.3 lists the time and effort invested in composing this thesis.

3.1 The Design and Creation Research Strategy

In this thesis, we aim to contribute knowledge to computer science and computer engineering. Therefore, we use the design and creation research strategy [38]. The core of this strategy is to develop a new IT product, commonly referred to as an artefact [39]. The definition of an artefact is quite broad. Such an artefact can be as simple as defining a new symbol or vocabulary, deriving new methods and models, or creating a program or prototype to solve a problem [40]. New knowledge and insights are derived by extensively analysing the created artefacts and critically evaluating their outputs. In contrast to product development, this research strategy provides explanations and justifications for the artefact.

3.2 Design and Creation of a Software-Based TSN End Station

As already introduced in this chapter, the design and creation research strategy aims to create new artefacts and derive new insights [41]. For instance, incorporating a new theory in an IT application can result in new insights during the usage or creation of the artefact. Within the scope of this thesis, we want to produce the artefact of a software-based TSN end station. Figure 3.1 depicts the artefacts we defined to produce and the knowledge outputs we expected.

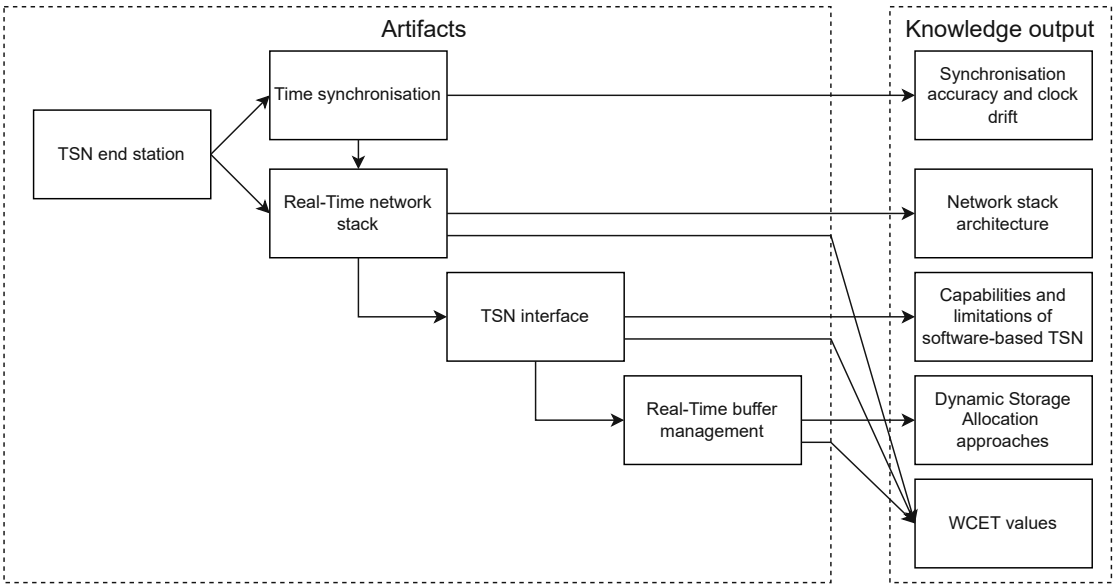


Figure 3.1: Artefact and knowledge output of this thesis

In Figure 3.1, on the left side, we have depicted the TSN end station, which is the overarching artefact we wanted to produce. This artefact requires time synchronisation and a RT network stack, where the former builds on the latter. From the time synchronisation, we want to get insights into the significance of the drift of our used clock and how accurate our defined time synchronisation is. The RT network stack requires a TSN-capable interface to provide TSN functionality. From the design and creation of this interface, we wanted to learn how capable a software implementation of TSN is and where the limitations lay. Due to the definition of TSN, a frame might get sent after a period of time. Hence it requires some buffer management. From this artefact, we retrieved knowledge about various approaches available and which are suitable for RTSs. In order to use our artefact of an RT-capable network stack in an RTS, it and all its sub-artefacts need to be WCET analysable.

The design and creation research strategy builds on various tools and development principles. The commonly used elements are awareness, suggestion, development, evaluation, and conclusion [42]. These principles are executed iteratively and revisited several times when new insights appear. Most work is part of the development phase of research focusing on computer-based systems. Therefore, this phase consists of the steps: analysis, design, implementation, and testing. Again these steps are executed iteratively with several executions of them. We used requirements imposed by underlying standards and software development best practices to derive our system design and implementation.

The thesis devotes a chapter dedicated to each artefact depicted in Figure 3.1. Each chapter consists of a section providing an introduction, the technical background, and related work of the covered topic. Further, each chapter has sections covering the design,

Topic	Task	Duration (days)
Time Predictable Network Stack	literature study	9
	system design and implementation	15
	evaluation	3
	sub-total	**27**
Time Synchronisation	literature study	6
	system design and implementation	6
	evaluation	3
	sub-total	**15**
Buffer Management	literature study	22
	system design and implementation	13
	evaluation	3
	sub-total	**38**
Time-Sensitive Networking	literature study	5
	system design and implementation	11
	evaluation	1
	sub-total	**17**
Thesis Document	finalisation	19
	total	**116 days**

Table 3.1: Estimated effort to create this thesis

implementation, and additional findings revealed during the development process. Based on the developed artefact, the single chapters provide a section evaluating the created artefact. Combining the results from the single evaluations, we provide a conclusion of the gathered knowledge and insights in Chapter 8.

3.3 Thesis Effort

In this section, we give an insight into the work conducted for this thesis. Table 3.1 lists the efforts broken down into single topics. To better track the time invested, we have split the efforts for each topic into three significant blocks.

- *Literature study*: gathering literature and published information about the given topic
- *System design and implementation*: deriving a design that is usable independent from the underlying use case of this thesis; create the code constructs derived from the system design
- *Evaluation*: creating an evaluation setup, conducting the evaluation, and assessing the results

CHAPTER 4

Time Predictable Network Stack Design

The network stack, sometimes referred to as the protocol stack, provides the interaction capabilities to collaborate with networks connected to a device or services via the Internet. Typically a layered architecture is used, where each layer has its purpose and works together with the neighbouring layers to allow communications between processes running on different devices. The tasks start at the bottom with access to the physical communication fabric. It continue with the translation of protocols and ends with the representation of information to the user. Given the more and more connected world, network stacks providing Internet Protocol (IP) connectivity to RTSs are common and need to adhere to certain RT requirements. In the following section, we focus on the overall architecture and design of an RT and WCET analysable network stack for an RTS.

Starting with Section 4.1, we introduce the different network layer models. Afterwards, we establish the individual layers with their most prominent protocols based on a generic network stack design in Section 4.2. With the different layers and protocols provided, we introduce our RT network stack architecture in Section 4.3. We used a static WCET analysis and runtime measurements to evaluate our proposed design. Section 4.5 provides the results of our measurements and interprets the gathered data. Finally, to wrap up this chapter, Section 4.6 summarises the chapter's contents.

4.1 Layered Network Architecture

There are multiple standards for defining computer networks. The majority follows one of two models, the *OSI reference model* or the so-called *Internet protocol suite*. The International Telecommunication Union Telecommunication Standardization Sector

(ITU-T) defines the OSI reference model, which is standardised inside the ISO/IEC 7498 part 1 [43]. It defines seven layers in ascending order, starting at the hardware and going up to the application. Protocols in higher layers mainly depend on protocols located in the layer beneath but never on a protocol in a higher layer.

1. *Physical layer*: It is closely connected to the hardware, taking care of the actual physical connection of devices.
2. *Data link layer*: Its purpose is to transmit data between nodes of a network segment. Other duties are handling physical layer access and detecting or even correcting errors. Protocols found in this layer are Ethernet and the Address Resolution Protocol (ARP).
3. *Network layer*: It takes care of forwarding and routing packets through a network and to other network segments. The most prominent members of this layer are the Internet Protocol (IP) and the Internet Control Message Protocol (ICMP).
4. *Transport layer*: It takes care of handling End-to-End (E2E) communication between applications using protocols like the Transmission Control Protocol (TCP) and the User Datagram Protocol (UDP).
5. *Session layer*: It handles the opening, closing, and managing sessions between end-user applications.
6. *Presentation layer*: It takes care of transforming data between different representations.
7. *Application layer*: It specifies the shared communication protocols and application interfaces. Members of this layer are protocols like the Domain Name System (DNS) and Secure Shell (SSH).

The OSI reference model is quite a detailed model. Therefore, implementations of network stacks commonly prefer simpler models. The other widely used model is the Internet protocol suite, often referenced as the TCP/IP stack or the *Internet reference model*. It represents the view on the Internet from the Internet Engineering Task Force (IETF), which is the driving force behind the Internet as we know it today. The Request for Comments (RFC) 1122 [44] and RFC 1123 [45] provide the definitions for this reference model. From the viewpoint of the IETF, the model consists of four layers:

1. *Link layer*: It handles lower-level protocols like Ethernet and handles medium access.
2. *Internet layer*: It handles routing and forwarding packets in local networks or to foreign networks.

3. *Transport layer*: This layer handles End-to-End (E2E) communication between processes.

4. *Application layer*: This layer covers the tasks of layers five to seven from the OSI reference model.

Figure 4.1 illustrates how the layers from the OSI reference model and the Internet protocol suite are related. The figure shows the layers from the OSI reference model on the left side and on the right, the corresponding layers from the Internet protocol suite. From hereon, we will mainly focus on the Internet protocol suite as our reference model, as it sufficiently describes the layers involved in the time-predicable network stack.

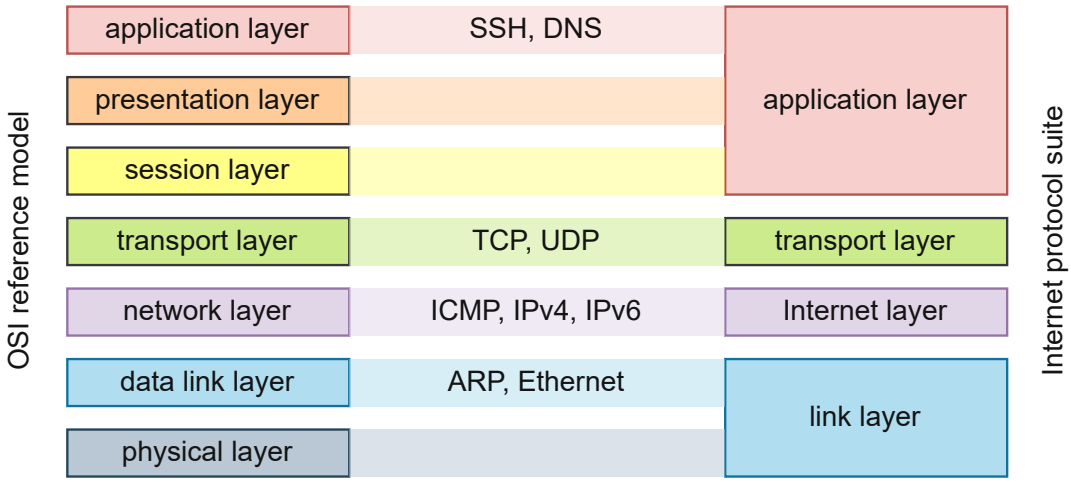


Figure 4.1: Comparison of the layers from the OSI reference model and the Internet protocol suite

4.2 A Generic Network Stack Architecture

In Section 4.1, we have introduced two referenced models of the layered network architecture. In this section, we want to outline an example of a generic network stack architecture based on the Internet protocol suite. Significant literature is available about the network stack of the Linux kernel, e.g., by Christian Benvenuti [46], which we used as input for this chapter.

In an Operating System (OS), the network stack allows communication with other entities, be it inside the device, another entity in the same network, or a service on the Internet. Therefore, each communicating application requires an interface to the network stack. Furthermore, the stack should provide a management interface for functionalities unrelated to communication, such as configuration and error handling. In addition, the stack has to interact with the communication channel to communicate with the outside world. Figure 4.2 depicts a generic network stack with the layers from the reference

model and necessary interfaces. The following sections elaborate further on the members of the individual layers and their interfaces.

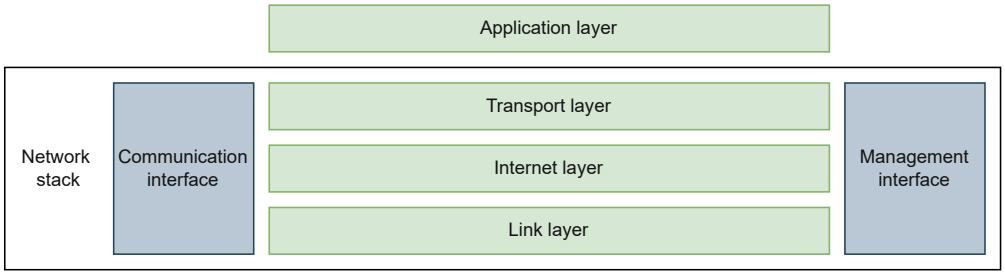


Figure 4.2: General architecture of a network stack

4.2.1 Link Layer

The link layer takes care of accessing a shared medium and handling communication via a single link. Most protocols for this layer use definitions based on the OSI reference model instead of the Internet protocol suite because it does not further specify access to a physical medium. The Internet protocol suite generally assumes that an underlying operational network infrastructure exists. Therefore, protocols like Ethernet and other IEEE protocols use the OSI reference model as a reference.

The main task of this layer is to handle point-to-point communication between two hosts connected to the same shared media. Protocols like Ethernet or Wireless Local Area Network (WLAN) provide this functionality. Upper-layer addressing is not applicable for this layer, so protocols like Address Resolution Protocol (ARP) query the corresponding link-layer addresses. A link-layer address is a property of the used communication hardware and can be considered immutable in the general case. In contrast, an upper layer address changes when the host changes to another network.

4.2.1.1 Ethernet

Ethernet is now the predominant technology used for wired networks – standardised first in 1981 as IEEE 802.3 [2]. One of the main reasons for its success was the very low price of a NIC. Another was that it was comparably easy to increase the transmission rates in contrast to competing technologies, starting at around 3 Mbit/s and now going up to 400 Gbit/s when using a medium like fibre optic cables.

Ethernet operates on a shared medium, called *the ether*, on which frames get exchanged. Therefore, it uses CSMA/CD to detect if two or more nodes try accessing the same communication medium simultaneously. Nowadays, most networks use switches, significantly reducing the probability of colliding messages. Still, the protocol has unreliable data transfer, as there is no data exchanged between the communication peers if a received frame is valid.

Ethernet uses the so-called hardware-dependent MAC addresses for addressing. Typically, the hardware provides these addresses, which generally cannot be modified. Nowadays, these addresses are called Extended Unique Identifiers (EUIs). These consist of six octets with a total length of 48 bits, typically represented in a hexadecimal format. No separator is defined to split the octets, but spaces, dashes and colons are often used. For example, a MAC address can look as follows: `12:34:56:78:9A:BC`.

The header of an Ethernet frame is quite simple, consisting only of the destination and source address and an identifier for the encapsulated protocol. In addition, the header may include an optional VLAN extension header. A VLAN separates a Local Area Network (LAN) into multiple logical networks. Ethernet uses a field called Frame Check Sequence (FCS) to detect corrupted frames. It is a 32-bit Cyclic Redundancy Check (CRC), which gets appended at the end of the frame.

An Ethernet frame has a minimum length of 64 bytes. If a frame is shorter than this, the sender shall append padding bytes to the data encapsulated in the frame. The maximum length of the data encapsulated in a frame is 1500 bytes. The Ethernet header has a length of 14 bytes but Ethernet extensions like VLAN can add additional headers. These extension headers do not affect the amount of user data encapsulated by Ethernet. For instance, an Ethernet frame can be tagged with up to two VLAN headers adding additional 8 bytes to the total length. In addition, Ethernet appends after the encapsulated data a 4 byte FCS. Therefore, an Ethernet frame is always between 64 bytes and 1526 bytes long.

The design for the required interfaces is pretty straightforward. Generally, we require one function for creating the header and one for reading the header. The communication hardware may not support the calculation of the FCS. Therefore, an implementation in software for the calculation may also be required. Additionally, a function for creating and interpreting the VLAN tags may be defined to simplify the process.

4.2.1.2 Address Resolution Protocol

The Address Resolution Protocol (ARP) queries the hardware address for a given protocol address and is standardised in RFC 826 [47]. The addresses get stored in an object called the ARP table. If a host does not find the corresponding hardware address inside this table, ARP sends a request to the connected network devices, asking who has the given protocol address. The connected devices either respond or forward the request to their neighbours. In practice, the only use case for ARP is querying the MAC address for a given Internet Protocol Version 4 (IPv4) address.

4.2.1.3 Driver interfaces

Now leaving the territory of protocols, the device drivers are the last step before entering the hardware domain. The development of such drivers is a well-known topic with freely accessible literature, e.g., by Jonathan Corbet, Alessandro Rubini, and Greg Kroah-

Hartman [48]. These software constructs provide an interface to the system for interacting with the hardware.

One of the tasks of a device driver is to expose the configuration options of the hardware to the system. Another task is to hand over data to the hardware to send out network frames. Additionally, the driver has to take over frames received by the communication hardware. Local buffers inside the device get used to store said frames. Therefore, a driver needs to perform at least a minimum amount of buffer management, whether inside the hardware or the (system's) memory. Further, the driver configures the network device via registers also located in the device, e.g., for setting the sending rate and selecting full or half duplex transmission. Such devices generally have special registers called Buffer Descriptors (BDs). The content of these registers defines the location of frames inside the device's memory and other vital data like length and possible transmission issues.

The interaction with the hardware is generally quite heterogeneous as different communication interfaces get used depending on the environment. Most systems use one of two methods to interact with the network interfaces. First, a network interface can be integrated into the chip or connected via an external memory bus. So-called Systems on a Chip (SoCs) typically employ the former method. Such systems commonly integrate a CPU and various Input/Output (IO) devices into a single chip. An external memory bus connects devices directly to the CPU core. Peripheral Component Interconnect Express (PCIe) is one of the broadest used memory buses for such a task. Both methods interact with the system's memory bus, which reads and writes to and from the device. This paradigm is called *memory-mapped IO* and allows µs to stay on an abstract level simplifying further explanations. As a third option, embedded devices that do not have an on-chip Ethernet MAC or PCIe support use buses like Inter-Integrated Circuit (I2C) and Serial Peripheral Interface (SPI) for interaction.

In the case of TSN, the Ethernet interface has some further requirements for accessing the shared medium. For instance, it requires the device to adhere to a schedule and stay synchronised with the network. A device may provide this functionality. If not, this has to be done by the device driver.

4.2.2 Internet Layer

We are continuing with the next higher layer in our architecture, the Internet layer. The primary protocol of this layer is the Internet Protocol (IP). Its primary purpose is to route packets from one host to another across one or more networks.

4.2.2.1 Internet Protocol

The Internet Protocol (IP) is the central protocol of the Internet protocol suite as one of its fundamental principles is *Everything over IP and IP over everything*. As already established, this protocol routes packet through a network and between networks. There are two major versions of IP, namely IPv4 and Internet Protocol Version 6 (IPv6). RFC

791 [49] defines IPv4, and RFC 2460 [50] first introduced IPv6, which was later obsoleted by RFC 8200 [51].

As the primary task of IP is to communicate between hosts, it uses an addressing format which is independent of the lower layers. Therefore, it employs so-called IP addresses to route packages between nodes. In IPv4, this address is written in dotted decimal format and is in total 32 Bit long, e.g., `123.45.67.89`. In IPv6, the address is extended to 128 Bit and written as colon-separated hexadecimal values, which are groups of four digits. To simplify these addresses, once a block of zeros can be shortened by using a double colon. Therefore, an IPv6 address can look as follows: `2001::1234:5678:9ABC:DEF0`.

The header of IPv4 provides the source and destination IP addresses for routing. The *protocol* field defines what upper layer protocol IP encapsulates in this packet, and for error detection, IP uses a 16-bit checksum. Additionally, the protocol header has a *time-to-live* field, which defines how many router hops a packet stays alive in a network. This field prevents packets from looping indefinitely in a network. The underlying communication channel may not allow arbitrarily long messages. Hence, IP takes care of fragmenting packets to fit into the Maximum Transmission Unit (MTU). Therefore, the *fragment offset* field defines the offset in bytes from the first frame for the reassembly of the packet.

Regarding the interface design for a network stack, two functions are sufficient for the implementation because of the stateless nature of IP. One function covers serialising the header, and another one covers the deserialising. The former sets the relevant fields and calculates the header checksum, whereas the latter has to validate the header and check for additional fragments.

4.2.2.2 Internet Control Message Protocol

RFC 792 [52] defines the Internet Control Message Protocol (ICMP) with its primary purpose of exchanging control messages between hosts. The most prominent function of this protocol is the *ping* functionality. Thereby, one node sends out an echo request and waits for another host to send a reply message. Another commonly used task of this protocol is to notify a host if a packet could not be delivered and why. ICMP datagrams get encapsulated in IP packets.

4.2.3 Transport Layer

The transport layer gets used for E2E communication between applications. In most, cases more than one application requires access to the network. Therefore, this layer needs to multiplex the network access. This multiplexing is done by using *ports* and allows the addressing of applications inside a host.

This layer provides the means for reliable data transfer, flow and congestion control. These functionalities require more complex services which are connection-oriented and have a stateful design. Protocols like TCP support this, but its implementation needs to

track the state of each of its connections. Nonetheless, there are also simpler protocols on this layer, like User Datagram Protocol (UDP), which is stateless and does not provide reliable data transfer.

4.2.3.1 User Datagram Protocol

The IETF defined the User Datagram Protocol (UDP) in RFC 768 [53]. UDP uses datagrams that provide connectionless communication, which means that no earlier interaction between two nodes is required to exchange data. Therefore, UDP does not provide reliable data transfer, as the protocol does not offer feedback to the sender about whether the message was received successfully. Additionally, it cannot guarantee that packages are not reordered or duplicated. Nonetheless, UDP uses a 16-bit checksum to check the received datagrams against corruption.

Many applications use UDP and accept the weaknesses like unreliable data transfer as it has less protocol overhead compared to protocols with reliable data transfer. Regarding the context of this thesis, RTSs commonly use UDP because of this property. In such cases, dropping packets is preferred over waiting for delayed retransmission, as the additional time required to receive the retransmitted packet may interfere with the remaining execution time of a task. In order to tolerate the dropping of packets, relevant data gets sent more frequently than actually required by the corresponding task.

Regarding the network stack's design, the interface for UDP can be held quite simply. In general, we require only two functions, one that creates the header and calculates the checksum and a second function deserialisation of the header and validation of the checksum. The inputs for the checksum calculation are the UDP header itself, the encapsulated user data, and the source and destination IP addresses.

4.2.3.2 Transmission Control Protocol

The Transmission Control Protocol (TCP) was defined by the IETF in RFC 793 [54]. In contrast to UDP, TCP is a connection-oriented protocol that requires establishing a connection between applications before exchanging data is possible. Therefore, TCP has the so-called *three-way handshake*, which defines how a client can connect to a server. Its purpose is to exchange sequence numbers to check which messages were received successfully. The three-way handshake consists of the following steps:

1. The client sets the `SYN` flag of the first message and transmits a random sequence number A to initiate the setup of a connection.
2. The server sets the `SYN-ACK` flag in its response message. Additionally, it sets the acknowledge value to $A+1$ and the sequence number to a random value B.
3. Finally, the client sets the `ACK` flag in its response message, the received acknowledgement number to $B+1$ and the sequence number to $A+1$.

The sequence number gets incremented after each transmission by the number of bytes transmitted. Its purpose is to determine if this message is in the correct order. Additionally, it indicates if this message is a duplicate or if a message is missing. Messages may be received out-of-order. Therefore, the TCP header includes the number of transmitted bytes to reconstruct the message. When a message is successfully received, the acknowledgement number is calculated by adding 1 to the received sequence number and sent back to the sender. This message then acknowledges all data received so far and informs the sender about the next expected sequence number by the receiver. A system can also acknowledge multiple messages at once. Therefore, the protocol uses the sequence number of the last successfully received message. If an incorrect message is detected, the acknowledge message shall take the sequence number of the last valid message before the invalid message. The sender then retransmits every message starting with the one after the last valid received message. The retransmission may create repeated transmissions of valid packets because an earlier one was not received or the header was invalid.

Additional features of TCP are flow and congestion control. The former handles that a receiver does not get overloaded with new messages, i.e., that the receive buffer does not overflow. The latter takes care of not overburdening the network itself. Typical measures used to implement are reducing the sending rate and limiting packets currently in transfer.

Regarding the interface design for a TCP implementation, we require at least two functions. One function is creating the header and calculating the checksum. The second function is validating received frames. In addition, we need to store data like sequence numbers and transmitted bytes for each connection to validate the reception of frames. Furthermore, the network stack has to store outgoing messages until it receives the acknowledgement. If the network stack does not receive the acknowledgement in time, a timeout mechanism has to retransmit the unacknowledged messages.

4.2.4 Application Layer

The next higher layer is the application layer, which is the top layer in our network model. It is home to various applications and services, like a web server or a file server. These applications commonly use application layer protocols which are out of the scope of this thesis. Nonetheless, these protocols depend on an interface to communicate, which has to be provided by the network stack. Therefore, the stack provides the communication interface, further described in Section 4.2.5.

4.2.5 Communication Interface

The network stack's Application Programming Interface (API) is the main point for interaction with the application layer. It provides the required interfaces for sending and receiving data to and from the connected networks. The main goal is to have a simple enough interface that is easily integrable but still provides enough freedom to work with most protocols. This is essential for integrating functionality into an application

or service. A commonly used implementation of such an interface is the socket API provided by the Linux kernel. Michael Kerrisk [55] published a book on how sockets work and their usage. The following explanations for the communication interface follow the concept of sockets.

The general procedure of transmitting a message consists of four steps:

1. get configuration data
2. set input data for protocols
3. perform protocol serialisations
4. hand the message over to the network interface

The second step takes the configuration data of the communication interface and the network stack, as well as the provided address, to generate the inputs required for serialising the individual protocol layers. Thereby, serialisation is the process of transforming data structures into an array of bytes. The main tasks handled in the serialisation are generating the protocol headers with the gathered information and calculating the protocol checksums. Finally, the buffers with the serialized protocol data and the user data are handed over to the communication interface driver.

When thinking of communication, we want to send out data and receive messages from the network. The reception process consists of the following four steps:

1. take over the frame from the network interface
2. deserialise the frame
3. execute header and user data checks of used protocols
4. provide the received data to the application

After a network interface has received a frame, the data is provided to the communication interface to start the deserialisation. The starting point of the procedure depends on the kind of hardware interface data, e.g., if the Ethernet NIC has received the frame, it is treated like an Ethernet frame. Most transmission protocols have a checksum or CRC to validate the correctness of the received data. If such a check fails, the received data is invalid, and the stack stops processing the frame. After each protocol layer, the communication interface checks whether there is a configuration which matches the deserialised header or not. If the deserialised header data matches a configuration, the communication interface forwards the encapsulated data to the corresponding application. Otherwise, the frame is either further processed by the next higher layer or discarded if the network stack does not support the protocol.

4.2.6 Management Interface

Figure 4.2 shows a large block on the right side called *Management Interface*. This entity handles the configuration of the individual layers and provides input data to the communication interface. One of the essential tasks of the network management is setting up the communication device drivers. Amongst other duties, this includes, if required, configuring hardware addresses and device capabilities.

Another vital task is to provide information about the used communication devices to the communication interface or other system services. As a system can have multiple network interfaces, the communication interface requires this information to generate the correct protocol headers. Depending on the interface, the single protocols need further information for serialisation and deserialisation.

An additional network management task is to initiate the deserialisation of frames received by the various network interfaces. This task is either done periodically or upon the reception of a new frame. Each frame is copied from the network interface into a temporary buffer and then provided to the communication interface to start the deserialisation. Additionally, it provides information about the kind of frame located in the buffer to the communication interface.

4.3 Real-Time Network Stack Architecture

Based on the interfaces and protocols introduced in the previous sections, we designed a software architecture for an RT-capable network stack, depicted in Figure 4.3. The individual layers already discussed in Section 4.1 are illustrated as green boxes. The grey box defines the boundaries of our design. Everything outside of the box is not part of our network stack.

At the bottom is the *link layer*, which is the layer with the most tasks. It provides protocols like Ethernet for media access and hosts the device drivers required to interact with the hardware. As there is a large set of communication channels, e.g., Bluetooth and WLAN, we restricted the *link layer* to Ethernet for this thesis. Further details on the tasks of the link layer got introduced in Section 4.2.1. Buffer management, which is essential for the operation of this layer, is used to provide storage for incoming and outgoing frames. Figure 4.3 depicts it on the right side next to the link layer. For further information on this topic, please have a look at Chapter 6. Additionally, on this layer are the driver interfaces introduced in Section 4.2.1.3. For an example of a TSN driver design and implementation, please look at Chapter 7.

The next higher layer in our network model is the *Internet layer*, which hosts the Internet Protocol (IP). In Section 4.2.2, we provide further details on this layer. As the top layer inside the network stack, the *transport layer* is home to the protocols used by most applications. Notable examples of such protocols are TCP and UDP, which were introduced in further detail in Section 4.2.3.

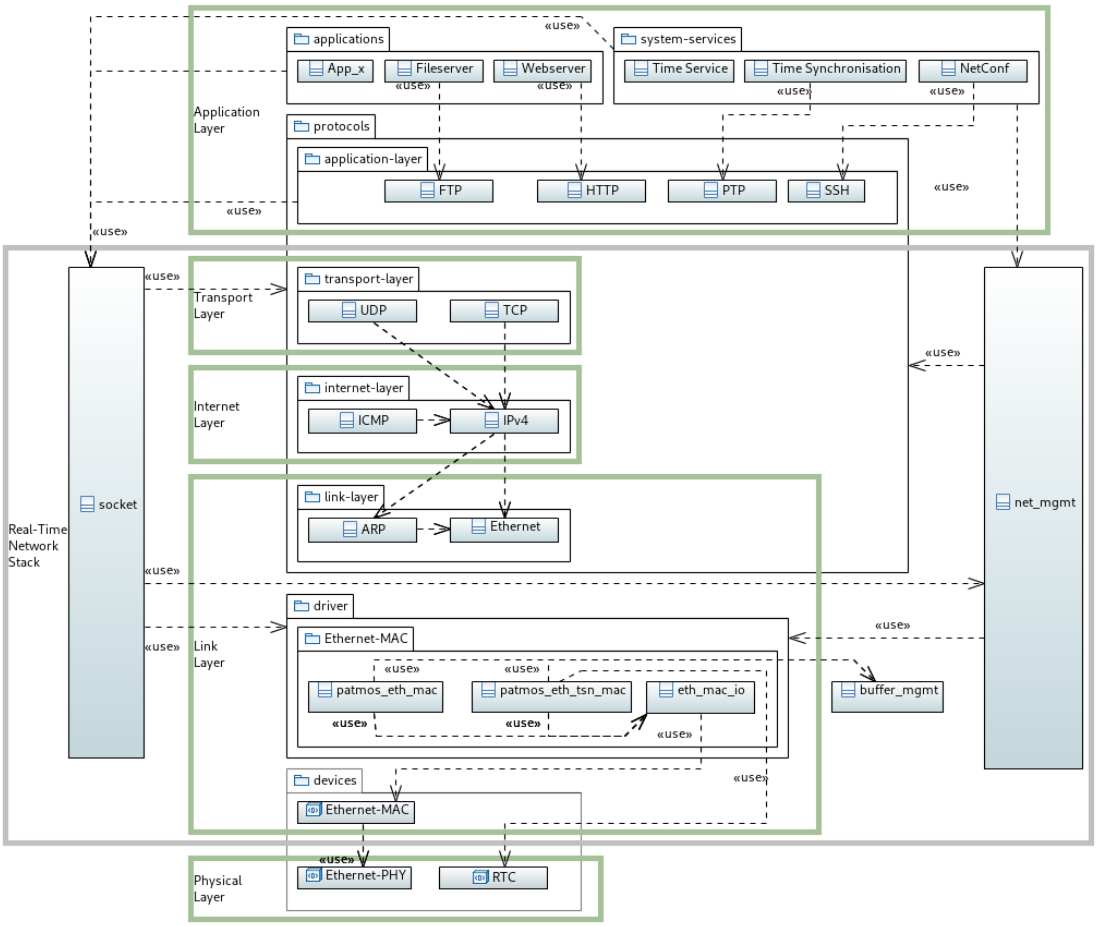


Figure 4.3: Overall architecture of the RT network stack

The generic applications illustrated at the top of Figure 4.3 exemplify how applications can interact with the network stack. These applications either use application layer protocols or directly interact with the network stack. Therefore, we defined our *socket* interface, which abstracts the inner workings of the network stack from the application layer. We decided to orientate the design on the principles of the sockets provided by the Linux kernel for two main reasons. Firstly, because it is a widely known and applied concept, which simplifies migrating existing applications to the proposed RT network stack architecture. Secondly, the interface has a simplistic design, which is a good starting point for integrating more complex protocols on top of it. An additional goal was to have the possibility to interact with the network from each layer, e.g., communicate directly via a link or Internet layer protocol. For further details on the application interface, please look at Section 4.2.5.

Section 4.2.6 states that a configuration and management interface is essential for a network stack. This interface acts not by interacting with the communication channels

but by changing the network stack's configuration parameters and the used hardware. General tasks like setting protocol addresses require this functionality so that the device is reachable from the network. In our design, we defined the *net_mgmt* module to handle these tasks.

4.4 Further Findings and Reflections

The scope of this thesis is limited, some protocol requirements were not satisfied, and we managed only to implement a subset of all protocol features. The stack has further limitations imposed by the constraints of the hardware used for this thesis. Therefore, we discuss the limitations of network stack implementation in this section.

4.4.1 ICMP, IPv6 and TCP Support

The focus of the network stack is RT communication which is not dependent on Internet Control Message Protocol (ICMP) support. The main focus of ICMP is the exchange of control messages, e.g., checking if a host or a port is reachable. In an RT network, the endpoints need to know their communication peers in advance, and there is typically no response to their messages. The additional exchange of ICMP messages would add load to the network, which is an undesired side effect.

This thesis focuses on already established automation networks, which primarily only support IPv4. Therefore, we excluded IPv6 from the thesis. Nonetheless, many topics discussed for IPv4 apply to IPv6 as well, with minor adaptations.

TCP sends out requests and waits for their responses which requires storing a state. The network stack design focuses on a stateless operation to be easily fully WCET analysable. Therefore, we excluded TCP from the implementation.

4.4.2 IP Fragmentation

One crucial part of packet-based communication is the transmission over different networks with varying MTUs. Switching from an Ethernet link to a connection with a lower MTU requires that frames are separated. In IPv4, this separation gets executed by each router hop if the MTU changes to a lower value. As we always assume an Ethernet MAC layer and no frames from a different network, this functionality is out of scope. Therefore, frames with a length greater than 1518 bytes get rejected by the stack and will not be transmitted. Received frames with a set fragmentation flag will be dropped on reception.

4.4.3 Hardware Offloading

Protocols like TCP and UDP use a 16-bit checksum to check if the received data is valid. Its calculation uses the whole user data and specific header fields and is a relatively simple iterative calculation. Nonetheless, many NICs have functions to offload this calculation

to free the host's CPU. As the used underlying Ethernet MAC does not provide such a feature, the checksum calculation got implemented in software.

The used hardware does not contain a so-called Direct Memory Access (DMA) controller. A DMA offloads the copy operations between memory locations and frees the CPU from this task. As Patmos does not have such a controller, the process gets executed in software, which is inefficient in comparison. Implementing these functions in hardware could significantly improve the performance of the network stack.

4.4.4 WCET Analysability Boundaries

As the RT network stack shall provide a generic implementation, some processes are hardly WCET analysable. For instance, an Ethernet frame has a variable length that ranges from 64 bytes up to 1518 bytes. Therefore, for each frame, the maximum length has to be assumed.

A more problematic boundary to determine is the number of received frames when extracting the data from the driver. We can determine a potential upper bound via the link speed. Nonetheless, the processor's speed may also limit this, i.e., how fast the driver can handle a frame. Another limiting factor is the memory space available inside the Ethernet MAC and the queue length of the received frames. Except for the queue length, the other parameters are hardware-dependent and require understanding the full hardware specifications to determine an upper bound. The WCET is further impacted by how often the stack executes the process of parsing frames. Such parameters get defined by the integrator integrating the stack into an application. I.e., reading the frames from the driver in a tighter interval reduces the number of frames stored in the device's memory but increases the load on the CPU.

If we start looking at the transmission of frames, the question resides about the boundary between the call to send and the actual frame sending. The process of handing over user data for header creation and finally transferring it to the driver can be analysed. If the driver is currently free, the transmission on the wire may start immediately, but if a frame is already waiting for transmission or the Ethernet MAC is currently busy, the frame gets queued. To determine an upper bound until the frame is transmitted, one has to know the maximum duration an Ethernet frame takes between starting and finishing a transmission. The transmission duration further depends on the link speed, the used Ethernet MAC and the PHY chip that handles the transmission over the wire.

4.4.5 WCET Analysability of the Reception

The first approach for handling the reception was to call a callback function when the frame was successfully parsed and then hand over the data. This approach has the downside that it uses function pointers, which get resolved during runtime. Therefore, a tool relying on a static code analysis cannot calculate the WCET. To allow WCET analysis of the entire reception of the frame, we decided to decouple the network stack operations from the receiving application. Therefore, the application has to provide

some memory space to the network stack to store buffers for the packets until the application reads them. This approach has the downside that the frame has to be copied multiple times, first from the memory space of the network interface to a temporary buffer inside the network stack's memory. Second, when the stack has identified the receiving application the data gets copied into the memory space from the application. Finally, the frame has to be copied a third time when the application reads out the message from the socket interface.

4.5 Evaluation

To evaluate the design of the proposed network stack, we decided to employ static WCET analysis to determine the worst-case scenarios and paths inside the network stack. We also decided to evaluate the average runtime it takes to hand over a packet to the network stack. This path includes creating the different headers for each layer and handing over the data to the Ethernet MAC driver.

4.5.1 Static WCET Analysis

Except for correct operation, the main focus is WCET analysability of the whole stack for usage in combination with RT applications. Since not all frames sent via the network are a maximum Ethernet frame of 1518 bytes, we decided also to analyse frames of a smaller size. We selected a frame length of 64 bytes, which is the minimum size of an Ethernet frame, since embedded systems tend to send shorter frames. Table 4.1 depicts the results of the static WCET analysis for both cases, 1518 and 64 bytes length. The WCET tooling used provides the worst-case runtime results in cycles. For better readability, we converted these values based on the clock frequency of our target system, which is 80 MHz. The table has two sections, the upper half represents the sending portion of the network stack, and the lower half represents the receiving portion.

When sending frames, we see that the most prominent contributors are the `network_send` and the `system_send` functions. Thereby, `network_send` handles handing over the just created frame to the driver interface. For more information on this, see Chapter 7. Next, `system_send` is a wrapper function for messages sent by the system and not meant for an application, e.g., sending ARP requests to retrieve the MAC address of the communication peer. Another significant contributor to the WCET is the `udp_build_datagram`, or rather its sub-function `udp_compute_checksum`. The system does not have any hardware accelerators to take over the calculation of the UDP checksum. Therefore, this has a significant impact on the WCET. As IP fragmentation is not supported, the loop boundary for the WCET analysis got derived from a maximum UDP payload size of 1472 bytes. As seen in Table 4.1, this still requires a decent amount of computation. In comparison, the checksum calculation for the IPv4 frame (`ipv4_compute_checksum`) has significantly less impact since IPv4 calculates the checksum only over the header and not the user data. Additionally, the function for building an Ethernet frame with or without the VLAN extension header is relatively fast. This is due to the usage of the hardware-accelerated

Function	WCET cycles	WCET at 80 MHz	64 bytes cycles	64 bytes at 80 MHz
socket_send ()	738914	9.24 ms	293255	3.67 ms
arp_build_request ()	4013	50.16 µs	4013	50.16 µs
system_send ()	286158	3.58 ms	107070	1.34 ms
network_send ()	272132	3.4 ms	93044	1.16 ms
udp_build_datagram ()	92781	1.16 ms	5298	66.23 µs
udp_compute_checksum ()	91399	1.14 ms	3916	48.95 µs
ipv4_build_datagram ()	7693	96.16 µs	7693	96.16 µs
ipv4_compute_checksum ()	2814	35.18 µs	2814	35.18 µs
ethernet_build_frame ()	2097	26.21 µs	2097	26.21 µs
ethernet_build_vlan_frame ()	2523	31.54 µs	2523	31.54 µs
network_send ()	272132	3.4 ms	93044	1.16 ms
socket_handle_frame ()	1246441	15.58 ms	515770	6.44 ms
socket_handle_mac_layer ()	772407	9.66 ms	450813	5.64 ms
ethernet_handle_frame ()	2590	32.38 µs	2590	32.38 µs
ethernet_handle_vlan_frame ()	582	7.28 µs	582	7.28 µs
arp_handle_message ()	65200	815 µs	65200	815 µs
system_send ()	286158	3.58 ms	107070	1.34 ms
socket_copy_to_receive_buffer ()	329488	4.12 ms	7894	98.68 µs
socket_handle_network_layer ()	18684	233.55 µs	18684	233.55 µs
ipv4_handle_packet ()	6013	75.16 µs	6013	75.16 µs
ipv4_verify_checksum ()	2556	31.95 µs	2556	31.95 µs
socket_handle_transport_layer ()	447678	5.6 ms	38601	482.51 µs
udp_handle_datagram ()	93328	1.17 ms	5845	73.06 µs
udp_verify_checksum ()	91468	1.14 ms	3985	49.81 µs
socket_copy_to_receive_buffer ()	329488	4.12 ms	7894	98.68 µs

Table 4.1: WCET analysis values for sending and receiving frames

CRC calculation provided by the Ethernet MAC. Therefore, the Ethernet-FCS calculation can be offloaded and reduces runtime.

Next, we take a look at the handling of received frames. We can see that the MAC layer and the network layer require significant time to process a frame. The most significant contributor is the copy function to the sockets receive buffer (`socket_copy_to_receive_buffer`). Significant improvements are achievable with better copy mechanisms. At the time of writing, the data is copied byte by byte, which is relatively inefficient. With a pretty optimistic assumption, an improved version can finish the process in about a quarter of the time used. Another significant contributor to the WCET is the verification of the UDP checksum, as it requires the same computational power as the checksum calculation. Again, the required computational time is significantly less for the IPv4 checksum. As the Ethernet MAC verifies Ethernet-FCS, we do not need to take care of this since erroneous frames get discarded by the hardware. For certain requests, e.g., ARP requests, the stack itself generates the appropriate response. Therefore, the `system_send` function also occurs in the receiving portion of the network stack.

4.5.2 Runtime Analysis

As already described, in addition to the worst case, we want to analyse the average case by measuring the time it takes to hand over a packet to the network stack. Therefore, we use two frame sizes, like for the WCET analysis. First, we create a packet that results in an Ethernet frame of maximum size. This maximum frame uses 14 bytes for the Ethernet header, 4 bytes for the FCS, and 1500 bytes of data. The available space gets filled with a UDP datagram which requires the UDP header (8 bytes) and the IPv4 header (20 bytes). In addition, we configured the VLAN tagging for the interface, which requires another 4 bytes for the extension header. Therefore, the remaining space (1472 bytes) was filled with a random string. We also ran the same test using an Ethernet frame of only 64 bytes with 18 bytes of UDP user data.

We executed our test 800 times and our measurements have shown that we have an approximately consistent runtime to execute `socket_send`. For sending an Ethernet frame with 64 bytes of data, we measured an average runtime of 829 µs, with deviations less than 2 µs. For the measurements with 1500 bytes of data, the average runtime is 1732 µs, with less than 2 µs of deviation.

4.6 Summary

In this chapter, we have introduced in Section 4.1 the layered design, which describes the communication patterns of networks. Based on these definitions, we described the various members of said layers from Section 4.2.1 to Section 4.2.4. Using these definitions, we designed an RT-capable network stack. We depicted the design in Section 4.3 and discussed its boundaries in Section 4.4. The network stack was then evaluated based on static WCET analysis and runtime measurements in Section 4.5. Based on these two sections, we have achieved our set goals, but there is still room for improvement in various sections. Be it by extending the stack with features of the Internet Protocol (IP) like fragmentation, the addition of protocols like ICMP and TCP, or optimising its general operation.

CHAPTER 5

Time Synchronisation

In almost any computer network, the individual devices have some local clock to keep track of the current time. Due to temperature variations, differences in voltage, imprecisely manufacturing of quartz crystals, and other physical constraints, these clocks will tick with slightly different frequencies and eventually drift apart. Time synchronisation, also commonly referred to as clock synchronisation, is employed to increase the accuracy of the system's time.

The chapter starts with introducing the topic of time synchronisation in Section 5.1. In Section 5.2, we introduce gPTP, which TSN employs to synchronise the communication peers. With gPTP introduced, in Section 5.3, we introduce the *time synchronisation service*, an application that provides time synchronisation. Rounding up the implementation, in Section 5.4, we provide delimitations, limitations and pitfalls encountered during design, implementation and usage. Further, we evaluated our time synchronisation service in Section 5.5. Lastly, in Section 5.6, we summarise the contents of this chapter.

5.1 Introduction to Time Synchronisation

To get a common understanding of this chapter, we introduce the following definitions:

- A *clock* is a device that ticks at a specific rate and modifies a counter value.
- The *time* is the counter value of the clock converted into seconds or any other time unit.
- A *timestamp* is a snapshot of the time.

In this section, we introduce synchronisation and the less commonly used syntonisation, see Section 5.1.1 To get a common notion of time across a network, various protocols

are in use. The most prominent protocols used are Network Time Protocol (NTP), see Section 5.1.2, and PTP, see Section 5.1.3.

5.1.1 Synchronisation and Syntonisation

The process of time synchronisation is the distribution of time to other devices. Its goal is that every synchronised device shall have the same notion of time. In the case of pure time synchronisation, if a new timestamp is received, the local clock's time will be adapted to this value, but the tick rate will stay the same. A protocol like NTP synchronises the time of computers in a network to a centralised time source.

Time syntonisation, on the other hand, means that two clocks shall have the same tick rate. In other words, e.g., a second shall be the same time interval in both clocks. To syntonise two clocks, the tick interval is constantly updated to achieve the same tick rate. Syntonisation does care about the actual time value inside the clock. Protocols like PTP use it in combination with time synchronisation to reach sub-microsecond accuracy, which control systems require.

5.1.2 Network Time Protocol

NTP is one of the most widely used protocols to distribute the current time in computer networks. It uses a client-server architecture on top of UDP to synchronise the computer to the Universal Time Coordinated (UTC). It provides accuracy within tens of milliseconds to the UTC over the Internet. In a LAN, this can improve to within a millisecond.

NTP uses the so-called *Stratum model*, where the system consists of multiple levels called *Stratum*. The root of this hierarchical structure is a set of time servers, which provide the time reference for the network. This level is called *Stratum 0*. These devices synchronise themselves to a so-called *primary time source*. This reference can be an atomic clock, a Global Navigation Satellite System (GNSS), e.g. Global Positioning System (GPS) or any other radio clock. The next layer is *Stratum 1*, which distributes the time information to other network devices. Members of this level are directly connected to a device from *Stratum 0* to get accurate time information for distribution. These computers are called primary time servers. *Stratum 1* devices distribute their time information to *Stratum 2* devices and so on up to *Stratum 15*. Additionally, servers of the same *Stratum* may also synchronise with each other for sanity checking and backup. The clients then synchronise their local clock by polling multiple time servers.

5.1.3 Precision Time Protocol

In general, time synchronisation in the range of milliseconds or tens of milliseconds is sufficient for most applications. For control systems or RTSs, this is different. There are higher degrees of synchronisation required. Therefore, protocols with higher accuracy, like PTP, which can provide sub-microsecond accuracy, get used. IEEE standardises it in IEEE 1588 [25]. The standard IEEE 802.1AS [26] defines the generalized Precision Time

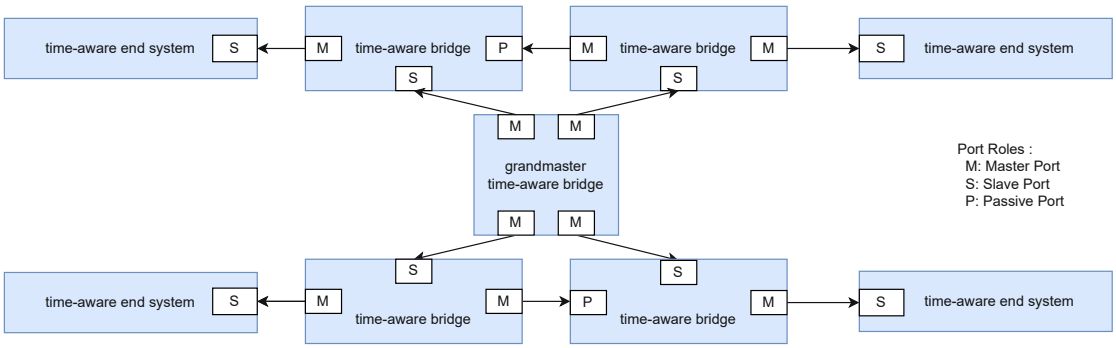


Figure 5.1: Example gPTP network

Protocol (gPTP). It uses a subset of PTP and can be seen as a configuration rather than a standalone protocol. In this thesis, we focus on TSN, which employs gPTP. Therefore, we provide a detailed introduction to gPTP.

5.2 The generalized Precision Time Protocol

PTP defines various ways to handle synchronisation, but gPTP only uses a subset. In PTP, various transport layers are available, but gPTP defines the use of Ethernet. Additionally, the Peer-to-Peer (P2P) delay mechanism shall be used instead of the E2E delay mechanism.

To understand the processes of time synchronisation in gPTP, we introduce the following terms:

time-aware system: a system that is aware that there exists a network time and has one or more PTP instances

grandmaster: the clock source of the network, which all members synchronise to

time-aware bridge: a time-aware system which distributes timing information to other time-aware systems and may also be the grandmaster

time-aware end station: a time-aware system, which either is the grandmaster of the network or a receiver of timing information

A time-aware end station can be either a complete system like a Microcontroller Unit (MCU) with a single interface that supports hardware synchronisation. It can also be a system with multiple NICs or a SoC with multiple Ethernet interfaces for redundant access. In such cases, each interface synchronises independently, establishing separate PTP instances. Nonetheless, only one interface shall be the source to synchronise the system's clock.

Figure 5.1 depicts an example gPTP network. The figure shows three types of ports: master, slave and passive. A *master port* is the time reference for a connection. If the connected peer is a *slave port*, then the time-aware system uses this port to synchronise the system. If the port is a *passive port*, it synchronises with the network but does not synchronize the system clock to the connected *master port*. Each time-aware system can only have one *slave port* as it can only synchronise to one master. In Figure 5.1, the grandmaster device is in the middle of the figure. The grandmaster is typically assigned using the *BMCA* based on factors like the local clock source quality and the location inside the network. The grandmaster provides the reference time for the whole network and has only *master ports*. In this example network, the time-aware bridges at the bottom and the top have redundant connections. These time-aware bridges will always have a *passive port* since the direct connection to the grandmaster is preferred.

5.2.1 Peer Delay Measurement

Figure 5.2 depicts an example of the P2P delay measurement. For simplicity reasons, we omit further correctional values of the messages transmitted. The single steps of the peer delay calculation are as follows:

1. The initiator sends a peer delay request message to a slave and stores the time *t1*.
2. The responder stores the time of reception as timestamp *t2*.
3. The responder sends a delay response message with the timestamp *t2* as its payload and stores the time of sending as *t3*.
4. The initiator stores the time of reception as timestamp *t4*.
5. The responder then sends a *follow-up* message which includes the timestamp *t3* as its payload.
6. The initiator can now calculate the path delay with the following equation: $meanPathDelay = \frac{(t4-t1)-(t3-t2)}{2}$.

5.2.2 Offset Calculation and Correction

In periodic intervals, the master port will send synchronisation messages to the slave port. Figure 5.3 depicts an example how this messages get exchanged. With the calculated path delay, the slave port can calculate the time offset from the master and adjust its local port clock.

1. The master port sends a *sync* message and stores the time as its *precise origin timestamp*.
2. When the slave receives the *sync* message, it creates the *sync ingress timestamp*.

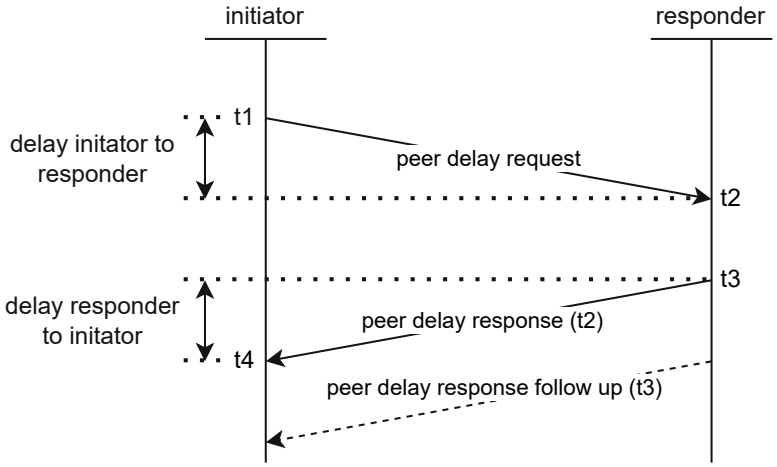


Figure 5.2: Peer delay measurement procedure, adapted from [26]

3. The master sends a *follow-up* message with its *precise origin timestamp* as the payload.

4. The slave can now calculate the offset and the corrected time at the master as follows:
$offset = syncIngressTimestamp - preciseOriginTimestamp - meanPathDelay$
$correctedMasterEventTimestamp = preciseOriginTimestamp + meanPathDelay.$

The slave then uses the calculated offset to adjust the local clock to syntonise with the master. The adaptation can be setting the time to the calculated corrected master time if the clocks have drifted too far away or the two clocks establish synchronisation for the first time. If this is not the case, the tick intervals of the clock shall be corrected. With this approach, it is possible to gain sub-microsecond accuracy between the clocks in a network.

5.2.3 Port State Decision Procedure

Figure 5.4 depicts the simplified port decision state machine for gPTP. This state machine is from PTP and has the configuration of gPTP applied to it, yielding a reduced diagram. The initial state *Power Up* indicates the power up of the system. It triggers the initialisation of the port in state *Initialising*. Per default, the port assumes the master role until a *State Decision Event* is triggered. Then the state machine switches into an intermediate state, depicted with a dashed outline. Section 5.2.4 explains the inner workings of the *State Decision Algorithm*, which provides a recommended Port State. That recommendation can either be *BMC Master* to switch into the master state, *BMC Slave* to act as a slave port or *BMC Passive* to enter passive operation. Additionally, the port can be disabled using a *Designated Disabled* event and enabled again via a *Designated Enabled* event.

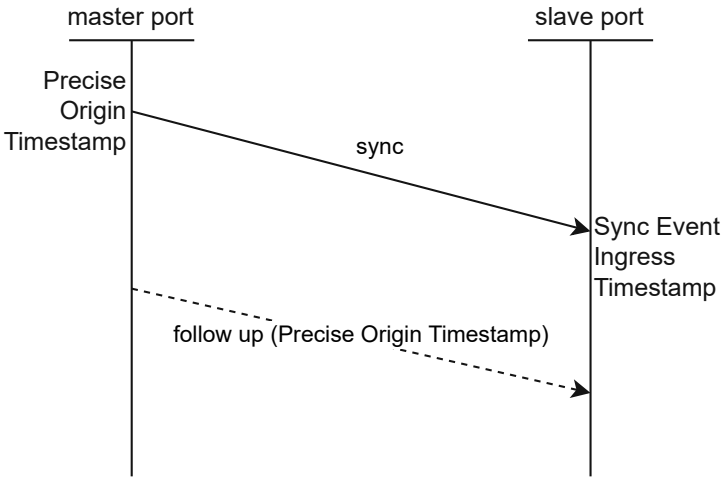


Figure 5.3: Synchronisation message exchange [26]

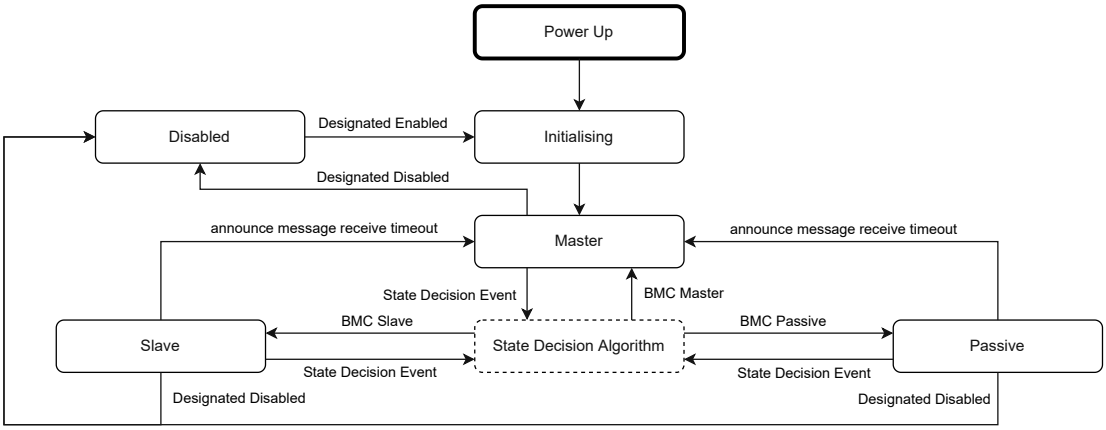


Figure 5.4: Port state decision state machine, adapted from [25]

5.2.4 Best Master Clock Algorithm

The BMCA is used inside gPTP networks to calculate what time-aware system shall be the *master clock*, commonly referred to as the grandmaster. Via this algorithm, gPTP creates a time synchronisation tree over a network. These algorithms are decision trees that operate on three different data sets. The first is the *Default Data Set (defaultDS)*, also referred to as *D0*, which is the data set of the system clock. Additionally, the data set E_{rbest} references the best external data set received on port r and the data set E_{best} is the best data set received on all ports. The following subsections introduce the State Decision AlgorithmState decision algorithm.

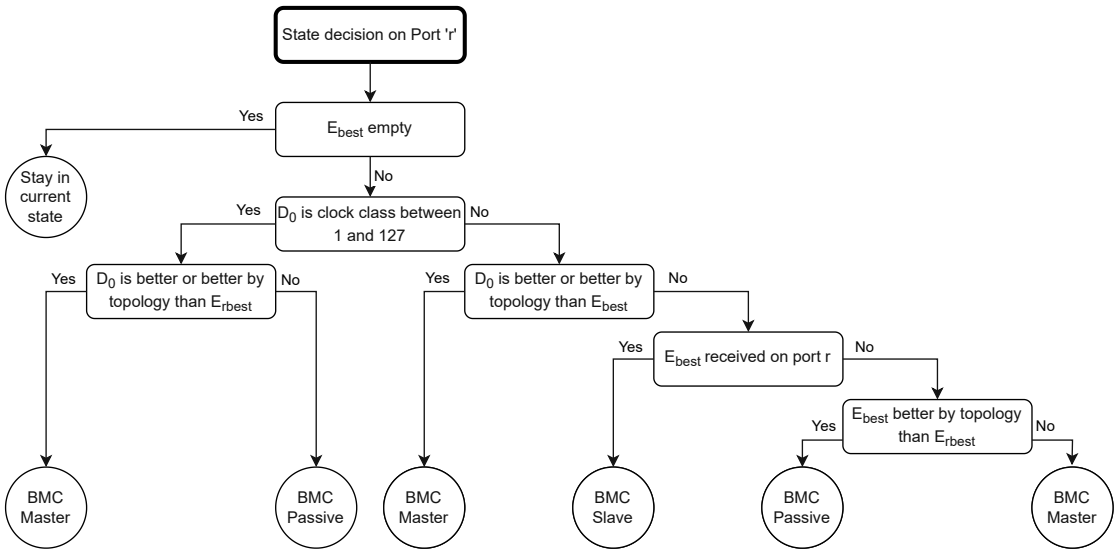


Figure 5.5: BMCA state decision algorithm, adapted from [25]

Figure 5.5 depicts the algorithm for the state decision. Ports distribute information about their local clock and their grandmaster using *announce messages*. If not defined otherwise in the configuration, each device sees itself as the grandmaster. Therefore, the algorithm checks first if the system has already received an announce message, i.e., if the system has already calculated an E_{best} data set. If this is still empty, the state shall not be changed, i.e., the system shall stay in its default state. The next step is to check if the port's clock has a class between 1 and 127, i.e., if its time source is a primary time reference, e.g., a GNSS. The grandmaster of a network is preferably a device with a high-accuracy time source. By definition, only one of these devices can be the grandmaster. Therefore, only one of these systems will synchronise with the network.

If the system's clock class is above 127, the decision algorithm compares the defaultDS with the best data set received on any port (E_{best}). Section 5.2.4.1 explains the algorithm to compare two data sets. If the defaultDS is better than E_{best}, then all ports will be configured as master ports, i.e., it is the grandmaster. If not, the port shall either receive timing information or distribute the timing information of another system.

Assuming port r received the data for the E_{best} data set, i.e., $E_{best} = E_{rbest}$, this port connects to the best master port. Then this port r shall set itself into the slave state and synchronise to the connected time-aware system. Given that another port received E_{best}, i.e., $E_{best} \neq E_{rbest}$, the algorithm has to decide if E_{best} is better than E_{rbest} or if it is only better by topology. If E_{best} is only better by topology, it indicates that both data sets have the same grandmaster, and only the path differs. Therefore, the port shall switch into the passive state. Given now that E_{best} is better than E_{rbest}, the port shall switch into the master state in order to distribute the timing information of E_{best}.

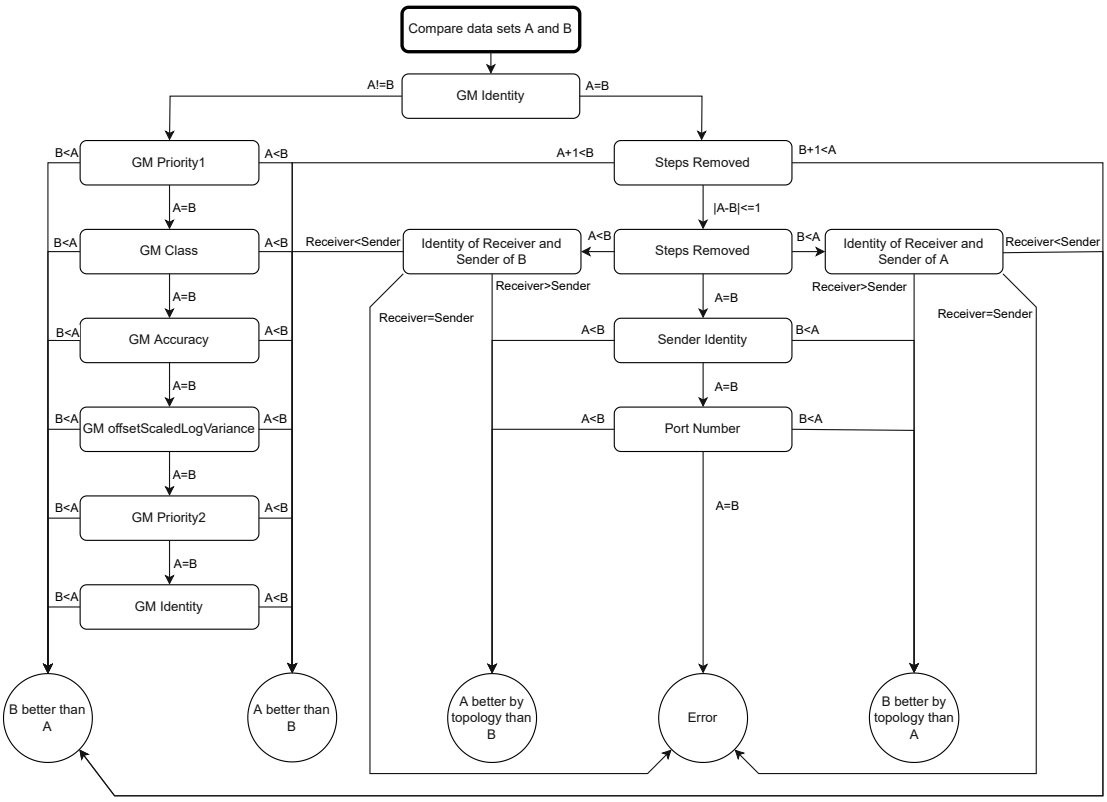


Figure 5.6: Data set comparison algorithm, adapted from [25]

5.2.4.1 Data Set Comparison Algorithm

Figure 5.6 shows the decision flow to compare two data sets. In general, this diagram consists of two halves. The left part compares two datasets with two grandmasters and determines which is the better data set. The right part compares two data sets with the same grandmaster and determines which data set has the better path to the grandmaster.

As already introduced, a PTP network typically has only a single grandmaster, but this is not true in some situations. For instance, when all the network members just started and all the devices see themselves as the grandmaster, or if the old grandmaster went offline. Therefore, a comparison algorithm is required to select the best grandmaster among two candidates. The comparisons use the members defined in Table 5.1.

Assuming that both data sets refer to the same grandmaster, the decision algorithm selects which path to select. Which path to select can be resolved via the *steps removed* value. If the difference between the number of steps is less or equal to one, it is unclear which path to select. In this case, the algorithm selects the path based on the lower identity values.

Properties used in Figure 5.6	Explanation
GM priority1 *GM priority2*	Priority values, lower values take precedence.
GM Identity *Identity of Sender* *Identity of Receiver*	A unique identifier for, clocks typically derived from the EUI 48 MAC address
GM class	Defines capabilities of a clock
GM Accuracy	Accuracy of the clock
GM offsetScaledLogVariance	An estimate of the clock drift in logarithmic scale
Steps Removed	How many hops the grandmaster is removed
Port Number of Receiver	The port number where the message was received

Table 5.1: Definitions of the values used for the data set decision algorithm.

5.3 Time Synchronisation Service

We decided to implement time synchronisation as a service which we will refer to as the time synchronisation service from now on. One of the two main reasons for this design decision was the location of gPTP, or rather PTP, inside the application layer. The other reason was that the network stack design has statelessness in its mind, and gPTP requires that states get stored internally.

Generally, a time-aware end stations could provide multiple Ethernet interfaces for redundant network access. Since this thesis does not focus on redundancy, we decided to design and implement the time synchronisation service only for a single interface. Therefore, the BMCA state decision algorithm from Figure 5.5 gets simplified as there is no difference between E_{rbest} and E_{best}. Nonetheless, the design allows easy extendability for redundancy. Therefore, one has to extend the update routine to query multiple sockets and extend the implemented design to be equivalent to Figure 5.5.

The time synchronisation service accesses the network interface driver to get and set data of the Real-Time Clock (RTC), e.g., the current time and message timestamps. The service uses this data internally to calculate offsets and mean path delays. With the calculated offset, the driver can manipulate the Ethernet MAC's RTC. This modification can be done by using the offset and letting the clock adjust its tick intervals. Alternatively, the service can also set the value of the RTC if the current time differs significantly.

As it is essential to synchronise with other devices on the network, applications may also require the current time. This use case may be especially relevant for TSN to allow the Real-Time Operating System (RTOS) to synchronise the execution of an application to the schedule. Therefore, the time synchronisation service provides an interface allowing applications to retrieve the current network time. Since gPTP communicates

its timestamps via a logical value in seconds and nanoseconds, the interface does not provide a conversion to a human-understandable wall clock format.

In addition to the system clock, each Ethernet port has its dedicated RTC, which can be adjusted to synchronise with another device. Therefore, two mechanisms were introduced, setting the time directly and configuring an offset calculated according to Section 5.2.2. The RTC takes an offset as input and tries to correct itself towards zero offset. Therefore, the size of the timer value increment is adapted based on the size of the offset, e.g., if the offset is positive, the RTC is running too fast. Therefore, the tick increment value gets reduced to slow the clock down. The inverse is applied if the offset value is negative, i.e., the clock is too slow. Kyriakakis et al. [35] implemented a hardware timestamping engine to improve the accuracy. This engine deserialises the RX and TX channels of the MII, which connects the Ethernet MAC to the PHY chip. The engine samples the MII channels and parses the received and transmitted frames. If the frame is a PTP frame, the engine issues a timestamp. This timestamping procedure is executed independently for RX and TX channels.

5.4 Further Findings and Reflections

In this section, we want to point out some findings we discovered during the implementation and testing of the time synchronisation service. For instance, we discovered that the used hardware implementation suffers from aliasing and that we require a qualification of timestamps.

5.4.1 Frame Sampling and Aliasing

The Ethernet MAC runs with an 80 MHz clock, but the MII has a specified clock for 100 Mbit/s of 25 MHz. Since the system's base clock is not a harmonic of the MII clock, effects like aliasing can occur when sampling the RX and TX channels. While analysing the provided hardware implementation, we have discovered that not all transitions get detected correctly, e.g., the hardware reads a bit as one, but its correct value is zero. This flawed detection mechanism impacts the operation of the time synchronisation service, as calculations cannot access the correct timestamp. Therefore, many frames and calculation results get discarded due to erroneous values.

This reduced generation of timestamps is inconvenient, but the time synchronisation still works. The downside is a negative impact on the time synchronisation service's accuracy. One can discard mean path delay values that exceed a maximum threshold, but this is not applicable regarding the offset calculation. A wrongly calculated offset can lead to a significant deviation between the two systems trying to synchronise.

5.4.2 Qualification of Timestamps

While testing and evaluating our proposed time synchronisation service, we discovered irregular values in the timestamping process. Seemingly at random, the calculated

offset between two timestamps differed by multiple milliseconds, in extreme cases going up to hundreds of milliseconds. Therefore, we evaluated the used implementations for fixed-point arithmetic and concluded that the issue either lies at the receiving or transmitting side. Given the assumption that we used *Linux PTP*, a well-known project that implements PTP, we assume that the root cause lies somewhere in the used hardware implementation.

We implemented a qualification mechanism to counter the identifiably wrong timestamps if a calculated offset shall be applied or discarded. If the last offset is within a certain threshold, e.g., 100 µs, and the newly calculated offset is also within this threshold, the time synchronisation service applies this new offset. The offset will not be applied if the last offset is within this threshold and the new offset exceeds the threshold. If only a single timestamp is wrong, this only discards the one wrong value, but this issue can occur subsequently. Therefore, we implemented a counter, which allows the acceptance of new offsets since the clock drift could exceed the threshold.

We use a simpler form of a fixed threshold to calculate the mean path delay. If the calculated value exceeds the threshold, we discard the value. In contrast to the offset offset, the mean path delay is not directly related to the master clock. Therefore, we decided not to implement a similar mechanism and only used the fixed threshold.

During testing we detected that the Ethernet link only operates in half duplex mode instead of full duplex. This can be backtracked to the Ethernet MAC of Patmos, which only supports 100 Mbit, but our grandmaster device only has a Gbit interface. Initially, we assumed that the half duplex mode causes increased collisions and bus arbitrations errors, hence Linux PTP may send incorrect timestamps. Further investigation using a 100 Mbit interface, which allowed full duplex operation, contradicted our theory.

5.5 Evaluation

Based on our use case, we implemented a test program which forces the device to act as a gPTP slave clock. For the master clock, we selected a Raspberry Pi Model 3b, which runs *Linux PTP*. We selected the Raspberry Pi as it provides a 100 Megabit (Mbit) interface, and the used Patmos Ethernet MAC only supports 100 Mbit. We used the gPTP configuration of Linux PTP, except the maximum propagation delay was adapted. This configuration specifies a *sync* message interval of 125 ms and a path delay calculation once every second. We focused our evaluation setup on how well the system stays synchronised with the master clock. Therefore, we did not further evaluate the mean path delay.

We configured Patmos to have two interfaces with PTP support to measure the significance of the clock drift. For the measurement, we synchronised one interface with a master clock and set the second interface's time once the first interface was synchronised. Given the assumption that the same implementation on the same silicon behaves equally, we can determine how far the local clock drifts away from the master clock. Figure 5.7

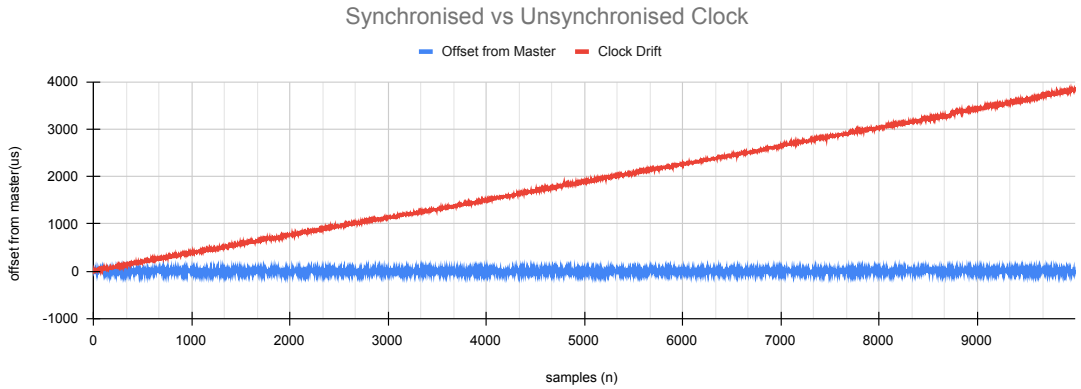


Figure 5.7: Measured clock drift

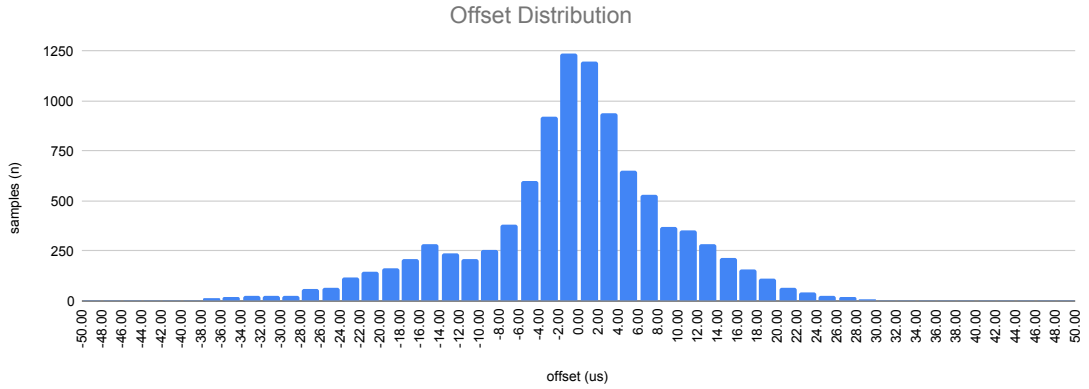


Figure 5.8: Offset from the master clock

depicts the results of our measurement. One can see that the measured clock drift over 10.000 samples is around 4 ms. Such a significant drift would invalidate the operation of a TSN schedule.

In Figure 5.8, we depict how well the slave clock stays synchronised. We see a cluster around the 0 µs mark. The outer edges of the offset are around +/- 40 µs and the standard deviation is 10.36 µs. There is still room for improvement as gPTP allows synchronisation in the sub-microsecond domain, but for this thesis, this is sufficient.

5.6 Summary

This chapter introduced the basics of time synchronisation and syntonisation and the most prominent protocols to synchronise devices on a network. Please have a look at Section 5.1 for more information. In Section 5.2, we gave a detailed introduction to gPTP, its delay measurement and decision algorithms. Based on these definitions, we

implemented the time synchronisation service, which we introduced in Section 5.3, with its limitations listed in Section 5.4. We evaluated the accuracy of our time synchronisation in Section 5.5.

CHAPTER 6

Buffer Management

Due to its indeterministic behaviour, DSA is rarely used in RTSs. Nonetheless, buffer management is one of the core elements of many network stacks as it temporarily stores incoming and outgoing frames. One of the most important roles is reserving the required memory space for a frame and ensuring safe access to it. There are many different management strategies with varying complexity, various pros and cons, which will be studied in this chapter.

In the following sections, we will take a closer look into different approaches for memory management, see Section 6.2, and algorithm types applying said approaches, see Section 6.3. The different types of algorithms will be compared to each other regarding worst case complexity and space requirements in Section 6.4. Finally, in Section 6.5, we present the design of our buffer management system and in Section 6.6 we discuss at the limitations of the presented design.

6.1 Introduction to Memory Management

Memory management is the process of overseeing the resource usage of a system's memory. This memory can either be the main memory, often referred to as system Random Access Memory (RAM) or the memory of a peripheral device, which needs some management. In general, its tasks are providing memory space dynamically to an application that requests it. Further, it cleans up the memory when the application returns it. For example, a client requires space to send a message to a server. Abraham Silberschatz et al. [56] introduce how OSs use memory management for concepts like virtual memory and paging.

Generally speaking, there are two types of memory management: manual and automatic. The former requires the application to request memory actively and free that memory again, whereas this is done automatically in the latter case. The runtime environment

does automatic memory management. For instance, it allocates memory on the stack to store variables when the program calls a subroutine. This memory is then released automatically when the routine is left again. In programming languages like Java, the runtime automatically allocates memory when the program initiates the creation of a new object. To clean up unused objects, the so-called garbage collector is used, which searches for such unused objects, and frees the memory space used by them.

In the remaining chapter, we focus on manual memory management, which can be split into static and DSA. The former defines that the memory is allocated in a static fashion, e.g., assigned at compile time. The latter defines that allocation is done dynamically during the program's runtime. For more information on the approaches to memory management see Section 6.2.

Finding free memory space is a job taken over by a so-called *Allocator*. It searches in the managed memory space for an unused area, which can be allocated for an application. One or more instances of *Free-Lists* typically handle the references to such accessible areas. Their task is to provide the allocator with a fast and efficient way to find a memory block with a matching size.

To correctly detect what portion of memory is actually free, many allocators use so-called *Boundary-Tags*. These tags are small data structures placed inside the managed memory either at the beginning and/or the end of a buffer. A Boundary-Tag typically has a reference to the start addresses of the next and the previous buffer as well as a reference to whether it is currently free or allocated.

Additionally, Boundary-Tags are useful helpers for *splitting and coalescing* buffers. If a free block of memory is larger than requested, it can be split into multiple chunks to improve memory utilisation. When the said block is freed again, the Boundary-Tags can be used to easily determine the boundaries of the neighbouring memory chunks and merge them together if they are also free. This is done to reduce fragmentation of the memory.

Fragmentation is a severe challenge regarding dynamic memory management. In general, it describes that the memory gets scattered into many distinct blocks. If the memory is not managed well, these scattered blocks limit the size of the largest chunk allocatable. In other words, there may be sufficient free memory in a system for a larger request, but this memory is not one continuous block and, therefore, cannot be allocated.

In Figure 6.1, the general structure of a memory management system is depicted. The *Allocator* spans over the whole memory, as it is the element which accesses the actual memory. The *Free-List* is a part of the allocator, as it stores the references to the free memory space. It points to *Boundary-Tags* which are not allocated. Said tags are located inside the memory and have references to their neighbouring memory spaces.

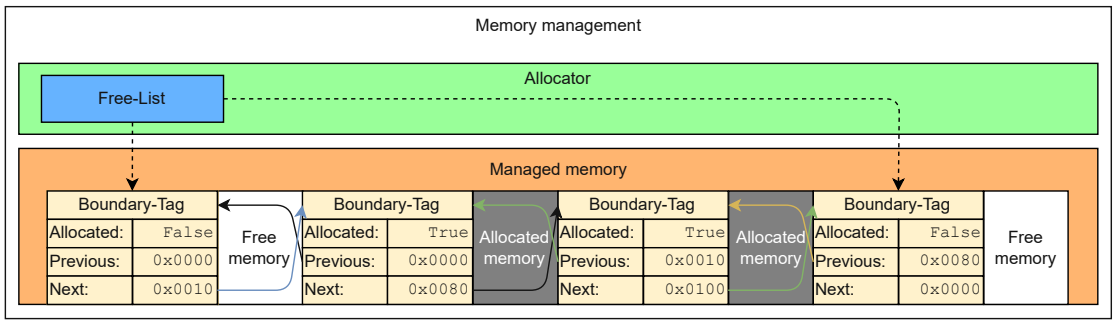


Figure 6.1: General overview of a memory management system

6.2 Manual Memory Management Approaches

There are various ways of managing memories. Typically, these mechanisms are split into static and dynamic memory allocation approaches. The former definition is quite narrow, but the latter defines a broad range of mechanisms. For the introduction to the topic of memory management, we introduce a more fine-grained separation of DSA approaches. These are further divided into buffers with uniform sizes, buffer classes with fixed and flexible sizes, and buffers with freely selectable sizes. The hierarchy of these algorithms is depicted in Figure 6.2. The further right a memory management approach is located, the more freedom of buffer sizes is provided, which generally indicates a more complex memory management algorithm but also improves utilisation.

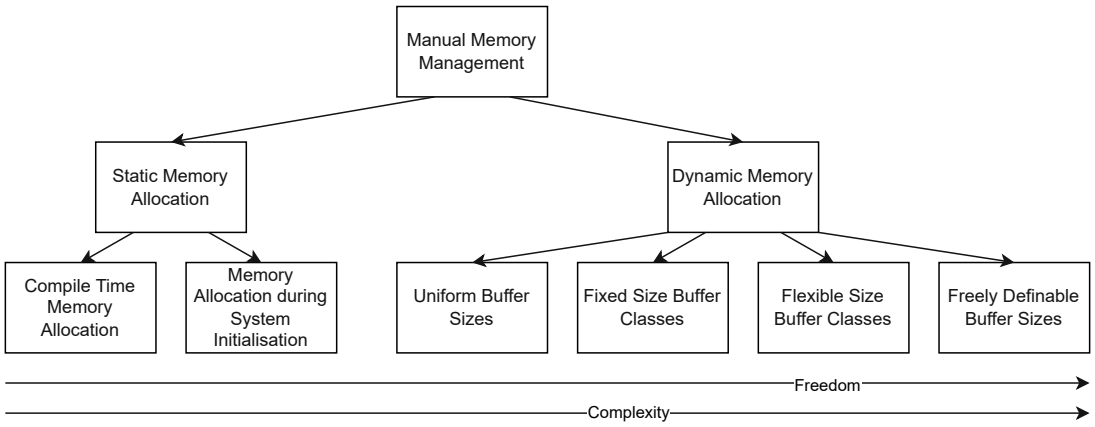


Figure 6.2: Different schemes of memory management mechanisms, extended from [57]

6.2.1 Static Memory Allocation

There are two common ways of statically reserving memory, either during compile time or during system initialisation. The former is done by defining elements in the program code, requiring the compiler to reserve memory space for them. In the simplest form, this is a global variable. The benefit of this method is that after the program compilation, one

knows how much memory the application requires, and no other memory management routine is necessary.

The second method, allocating memory space during initialisation, is often used when the system has multiple modes of operation that may have different applications to run. This can lead to a smaller memory footprint, as the system does not need to allocate memory for applications that are not being executed. As this is handled during the start-up procedures and no further dynamic memory allocation takes place, one can consider allocated memory as static. Furthermore, as the procedure is executed during the initialisation phase, the buffer management commonly does not need to adhere to RT requirements.

Additionally to the lower overhead, these two methods are not impacted by fragmented memory areas, i.e., the memory space used does not need to be continuous or the same type of memory. Nonetheless, reserving buffers for all applications can impact lower-end systems, as insufficient memory might be available.

6.2.2 Dynamic Memory Allocation

As the name already implies, dynamic memory allocation or Dynamic Storage Allocation (DSA) allocates memory space dynamically during runtime. The size of the buffers can be fixed, a set of different values, or completely free. The primary benefit of dynamic over static memory management is that applications can reserve memory temporarily and free it again, reducing the required memory in general. Nonetheless, it requires more complex approaches and may introduce indeterminism.

6.2.2.1 Blocks with Uniform Size

The simplest form of a DSA approach is to split the managed memory into uniform blocks or chunks with a fixed size and dynamically distribute them. An algorithm of this class must check if a buffer is available to allocate a buffer. No searching is required as all buffers have the same fixed size. This approach is a quite basic form of dynamic memory allocation and can be implemented with quite simple data structures reducing the total complexity.

The downsides of such algorithms come from the simplicity itself. Because all buffers are the same size, much unusable memory will be created when the buffer is significantly larger than the requested size. This unusable memory has been allocated to a buffer but is not used by the application which allocated it. Therefore, it cannot be used for any other application.

6.2.2.2 Blocks with Different Sizes

The next step is to use multiple classes of buffers, to improve on the issues of uniform buffers. Each class defines a certain fixed block size, and the managed memory space is

split up so that each class has at least one buffer exists. These fixed-size classes improve memory space utilisation and reduce the amount of unusable space created.

This additional task comes at the cost of more complex management algorithms to handle the memory space and manage the buffers assigned for the given classes. These DSA algorithms generally want to return a buffer as close as possible to the requested size. Therefore, the algorithm must first find a suitable size buffer class and then find a free buffer of said class or of a bigger class.

The steps introduced by this class greatly improve utilisation and reduce the amount of unusable space. Nonetheless, if the matching class has no free buffer left, the next larger free buffer has to be used. This again creates unusable space, which shall be avoided.

6.2.2.3 Blocks with Flexible Size

To improve further on the issue of unusable blocks, the next step, which can be taken, is to split up memory at runtime. In this case, the managed memory space is split up into blocks of the largest class. When a buffer is requested, the algorithm searches for a class with free buffers of at least the requested size and allocates it. If the buffer size is larger than the requested size, the buffer will be split into buffers of smaller classes.

This category has less added complexity in comparison to the fixed-size blocks as the used data structures can largely stay the same. The only additional step is the splitting and coalescing of buffers. This small addition largely improves utilisation and greatly reduces the amount of unusable space, but it is highly dependent on the defined classes. If these buffer classes are not well selected, there can still be quite large blocks of unusable memory, or the number of buffers can explode. This can happen if there is a large gap between two buffer classes, e.g., having the classes 100 and 1000, requesting a buffer of size 100 generates 10 buffers.

6.2.2.4 Blocks with Freely Selectable Sizes

Freely selectable buffer sizes can be used instead of predefined classes for further utilisation improvements. In this case, the managed memory space will be defined as one large buffer and every time a buffer shall be allocated, the requested amount is taken from that buffer. In general, this now provides the means of having ideal memory utilisation and no unusable memory.

Nonetheless, an algorithm for this approach may be pretty complex without classes or any predefined size. Such an approach requires a data structure which can efficiently manage the various class sizes. When a buffer is requested, the algorithm may search through the whole memory space to find a matching buffer. This can lead to quite a long runtime for allocating and freeing buffers.

6.3 Dynamic Storage Allocation Algorithm Types

Now, we look closely at types of algorithms based on the approaches introduced in Section 6.2. Puaut [58] and Awais et al. [57] investigated a set of commonly used DSA algorithms regarding worst case complexity and Worst-Case Execution Time (WCET). In the following sections, common algorithm types as well as their benefits and drawbacks are discussed. Furthermore, at least one commonly used algorithm of each type are presented. Additionally, we sketch the worst cases scenarios, and give a rough estimation for worst case complexity and space allocation.

6.3.1 Sequential Fits

Simple algorithms and data structures are sufficient to realise sequential fit buffer allocation strategies. As the name implies, searching for a matching buffer is done by iterating sequentially over a so-called Free-List, which holds all free buffers. The following four strategies are commonly employed:

- *First Fit*: returns the first buffer with sufficient size
- *Next Fit*: an extension to first fit that searches for a buffer with sufficient size in the Free-List, starting after the position of the last allocated buffer
- *Best Fit*: tries to find the best fitting buffer to the requested size, with as little overhead as possible
- *Worst Fit*: tries to find the largest available memory block

Figure 6.3 depicts an example for allocating a buffer of 16 bytes. The double-ended arrows indicate the doubly linked list relation between the individual buffers. The arrows on top depict which buffer is returned depending on the strategy. Given the *First Fit* strategy, the first buffer is taken, which has a size of 20 bytes. If the *Worst Fit* strategy is used, the largest available buffer is taken, which has a size of 48 bytes in this case. Using the *Best Fit* strategy, the first buffer with 16 bytes is used as it fits exactly the requested size. Assuming the *Next Fit* strategy is used, the dashed arrow indicates the last returned buffer element. Therefore, the algorithm returns the other free 16 bytes buffer.

Sequential fits are relatively simple, but the worst case complexity depends on the number of buffers in the Free-List. The worst case for allocating a buffer would be if the managed memory alternates between busy and free buffers of minimum size $B_{\min}$. Therefore, the worst case complexity is $O(n)$ where n is the length of the Free-List, which is not constant. The maximum length of the Free-List can be calculated as follows $n = \frac{M}{2*B_{\min}}$, where M is the total size of the memory and $B_{\min}$ the minimum buffer size. Therefore, the maximum length of the Free-List grows with the available memory space, yielding $O(M)$.

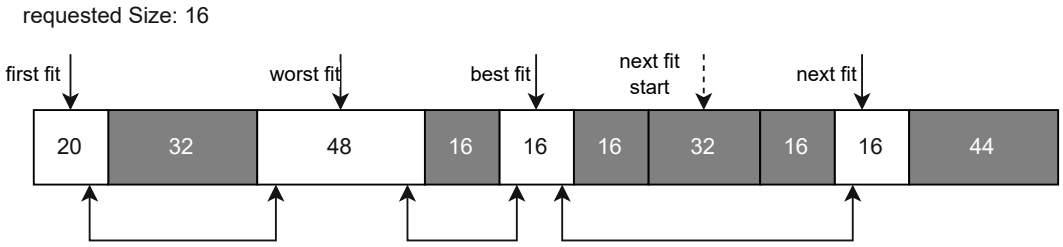


Figure 6.3: General structure of a sequential fit

In contrast, freeing a buffer is as simple as adding an entry to a queue, which can be done in constant time $O(1)$. The worst case for deallocating a buffer is when its neighbouring blocks are free. Given the assumption of immediate merging, up to three blocks may be coalesced after freeing a buffer.

The size of a linked list grows linearly based on the number of elements stored in it. Therefore, the size of the Free-List also grows linearly based on the number of free buffers. Since the size of the memory limits the number of free buffers, we can derive that the required space grows with $O(\frac{M}{2*B_{\min}})$, which can be simplified to $O(M)$.

6.3.2 Bitmap Fits

As the name already implies, bitmap fits use bitmaps to keep track of which portion of the managed memory is already allocated. One of the key benefits of this type of algorithm is that the information is encoded in small pieces, typically a vector with 32 or 64 boolean values. In other words, a single integer value. This reduces the probability of having a cache miss and, therefore, improves the general responsiveness of this class of algorithms. With a simple bitwise operation, one reads out which buffer classes have buffers available or which do not.

6.3.3 Segregated Fits and Indexed Fits

Segregated fit algorithms use independent data structures for the different buffer classes, e.g., having a linked list for each buffer size. The blocks of the separate Free-Lists are only segregated logically but not physically.

The indexed fits can be seen as an extension to the segregated fits, as they use more complex data structures, such as trees, to manage the several Free-Lists. The tree nodes represent the different buffer classes, and each node is linked to a Free-List managing the free buffers. Common algorithms for the indexed fit class are:

- *Ordered binary tree best-fit*: As the name already implies, the base data structure is a binary tree, which is sorted based on the size of the buffer class. Each node has a linked list storing the free buffer entries, which are removed when a buffer is allocated and added again when it is freed.

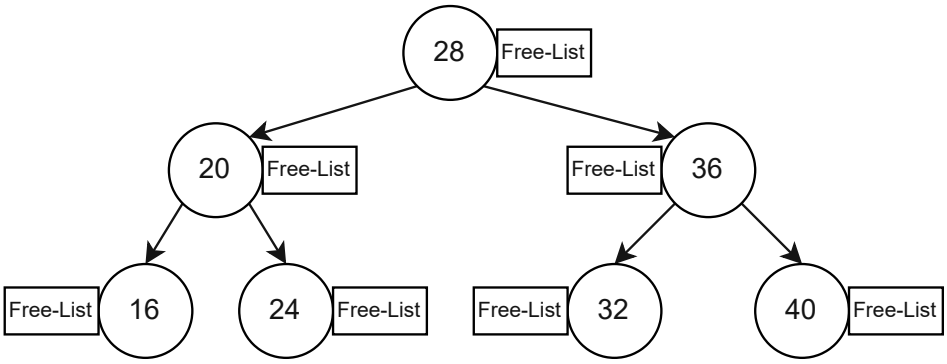


Figure 6.4: General structure of a binary tree indexed fit

- *Fast-fit* [59]: This variant uses a Cartesian tree as the underlying data structure, with the buffer address as its primary key and the size of the buffer as its secondary key. Acquiring a buffer is done by traversing the tree until a matching buffer size is found. The primary key is used when a block is freed again.

Figure 6.4 depicts the underlying data structure of the *ordered binary tree best-fit* algorithm. Each node represents a class of buffers and has a Free-List associated with it. When a buffer is requested, the tree is traversed to find the matching node and an element is removed from the associated Free-List.

In the following paragraphs, we will take a look at the worst case behaviour of the *ordered binary tree best-fit* algorithm. Initially, we assume constant class sizes. Therefore, the tree is a predefined structure and it can be assumed that it is balanced. The worst case for allocating a buffer is when the requested size belongs to the smallest class and only buffers of the largest class are available. Assuming that an inner node has no knowledge of the Free-Lists of its child classes, the algorithm may be required to check all k classes to find a free buffer. This results in a worst case complexity of $O(k)$.

The algorithm can be optimised by providing information of its children to each inner node, which reduces the worst case complexity to $O(\log_2(k))$. Freeing a buffer has the same worst case as a requesting a buffer, as the same steps have to be taken, i.e., locating the correct buffer class. Given the constant class sizes, freeing a buffer again, i.e., inserting an element into the tree, has also a worst case complexity of $O(\log_2(k))$.

Extending the current case with splitting and merging, the worst case complexity is not impacted significantly. After a buffer has been split, the newly created buffer has to be added to a Free-List, which has the same worst case complexity as finding the corresponding buffer class for allocating a buffer. The same is applicable for freeing a buffer. As merging a buffer and removing an entry from a doubly linked list can be done in $O(1)$, this does not effect the worst case complexity. Therefore, the worst case complexity is still $O(\log_2(k))$.

We are now extending the algorithm to define buffer classes during runtime. Hence, we cannot use a predefined tree, and one can no longer assume a balanced tree. Therefore, in the worst case, the tree leans entirely to one side; in other words, it forms a linked list. The worst case for requesting or freeing a buffer is when the buffer size belongs to the leaf of the tree. Accessing a leaf node requires the traversal of the whole tree up to n elements. This results in a worst case complexity of $O(n)$. Nonetheless, this case can be improved by using self-balancing binary search trees such as a red-black tree by Bayer [60], which has a worst case complexity of $O(\log(n))$ for finding, inserting and removing an element.

Given that the buffer count n is not a constant, we take a closer look at its upper bound. We define the smallest buffer as B_{min} and M for the size of the managed memory. To generate the maximum tree, M needs to be split into distinct values starting with B_{min} up to M, where a buffer of minimum size separates each buffer. This can be written as Equation 6.1 [1] and solved for the maximum number of free buffers n, as seen in Equation 6.2 [2].

$$M = \sum_{B_{\text{min}}}^{n} i + B_{\text{min}} = -\frac{1}{2}(B_{\text{min}} - n - 1)(3 * B_{\text{min}} + n) \tag{6.1}$$

$$n = \frac{1}{2}(\sqrt{16B_{\text{min}}^2 - 8B_{\text{min}} + 8M + 1} - 2 * B_{\text{min}} - 1) \tag{6.2}$$

One can assume that the smallest buffer is in the region of a couple of bytes and the size of the managed memory is in the range of multiple kilobytes, hence $B_{\text{min}} \ll M$. Therefore, n is bounded with the available memory size $O(\sqrt{M})$. Inserting this bound into the worst case search time of the self balancing binary tree yields $O(\log(\sqrt{M}))$.

We are now taking a look at the space requirements for the underlying data structure. As each node corresponds to a buffer class which holds a Free-List, two cases need to be considered: the first case for having a tree of maximum size and the second case for having a linked list of maximum length. Both structures, the binary tree and the linked list, grow linearly with the number of elements stored n. Given the calculation for the maximum size of the tree from Equation 6.2, this grows with $O(\sqrt{M})$. Taking a look at the latter case, the Free-List has maximum length, when the memory consists of buffers with minimum size alternating between free and allocated state. Therefore, the maximum length of the Free-List is $n = \frac{M}{2*B_{\text{min}}}$, which grows linearly with $O(n)$ and, therefore, faster than $O(\sqrt{M})$.

[1] https://www.wolframalpha.com/input?i2d=true&i=M%3Dsum+i+%2Bb%5C%2844%29+i%3Db+to+n, accessed 20.4.2022

[2] https://www.wolframalpha.com/input?i=M+%3D+-1%2F2+%28b+-+n+-+1%29+%283+b+%2B+n%29%2C+solve+for+n, negative sign ignored as only a positive count of buffers exists, accessed 20.4.2022

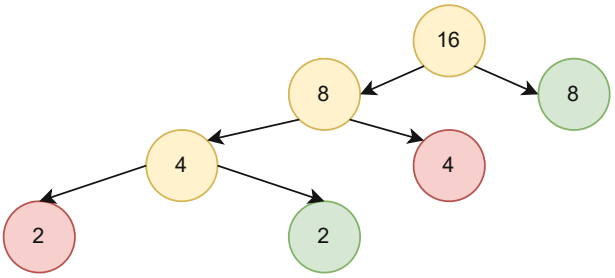


Figure 6.5: General structure of a binary buddy system

6.3.4 Buddy Systems

The buddy system algorithm type was first presented by Knowlton [61] and uses recursive subdivision to calculate the buffer sizes. First, a the memory area is split into buffers of maximum size, which all build the root of a tree. When a new buffer shall be allocated, the size is rounded up to the next possible block size. The Free-Lists will be checked for a buffer of the matching size. If the smallest available buffer is larger than the requested size, the buffer will be split according to the underlying policy. The newly created nodes are called buddies or mates and one of these will be added to the corresponding Free-List. The other one is either returned as buffer or subdivided further until matching the size for the requested buffer. When a node is freed again, it can only be merged with its buddy and only if none of the buddy's children are allocated. Therefore, the buddy system suffers a lot from fragmentation, creating up to 50% unused blocks when using a binary tree as the base data structure.

An example for the structure of a binary buddy tree is depicted in Figure 6.5, where red nodes depict allocated buffers, green nodes free buffers, and yellow nodes are entries, where at least one child is allocated. Given that a buffer of size 4 is requested, the currently free node of size 8 is split into 2 nodes. If the buffer of size 4 on the left side of the tree is freed again, it cannot be merged as its buddy has at least one allocated child.

The amount of nodes created at each step of the tree depends on the size of the memory, the size of the buffer entries, and the concrete implementation of the algorithm itself. In general, for all realisations, the worst case of allocating a buffer, is when there are only entries of the maximum size and the requested space is of minimum size. The following argumentations will assume the binary buddy system, where all buffer sizes are powers of two.

When a buffer is requested and an existing buffer has to be split, one buddy is added to the total number of buffers, and the other buddy will be split further if required. This is repeated $k-1$ times and creates k buffer entries, which is equal to the number of buffer classes and the height of the tree. As all classes are a power of two apart, a tree can always be split into the minimum buffer. Given now that B_{min} is the smallest buffer and B_{max} is the largest buffer, one can calculate the number of leafs by $l = \frac{B_{\text{max}}}{B_{\text{min}}}$. The tree can be maximised if the maximum buffer B_{max} is set to the size of the memory M

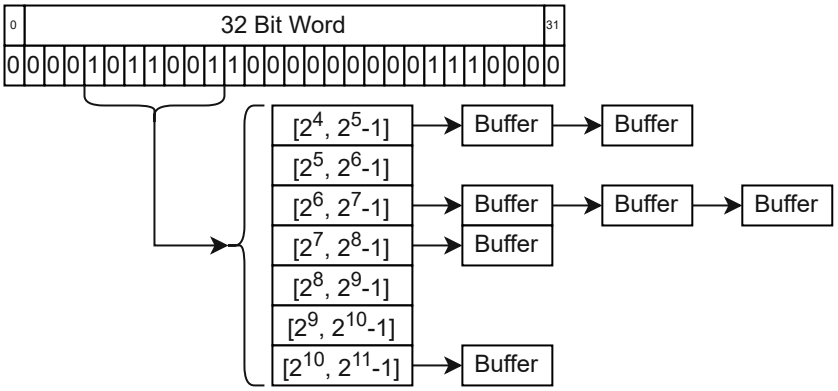


Figure 6.6: General architecture of Half-Fit, adapted from [62]

yielding $l = \frac{M}{B_{\min}}$. Based on this, the maximum number of classes are $k = \log_2(\frac{M}{B_{\min}}) + 1$, and therefore, the allocation of a buffer grows with $O(\log_2(\frac{M}{B_{\min}}))$. If a buffer of minimum size is freed, it may be merged with up to k other buddies. Therefore, the deallocation has also a worst complexity of $O(\log_2(\frac{M}{B_{\min}}))$.

The tree for the buddy system does not only need to store free elements, but also allocated ones. The worst case regarding the spacial allocation is if the allocated buffers are all of minimal size. With each new layer in a binary tree, the number of new leafs is the double of the previous leafs of the last layer. Therefore, the total number of nodes inside the tree can be calculated with $n = 2 * \frac{M}{B_{\min}} - 1$ and the required space grows with $O(\frac{M}{B_{\min}})$.

6.3.5 Hybrid Allocators

Commonly, a single type of algorithm does not satisfy all requirements. Therefore, many new algorithms are a hybrid out of at least two types. In this section we will look at *Half-Fit* and TLSF, which have a constant worst case complexity. These algorithms use concepts from the segregated and bitmapped fit to achieve this.

6.3.5.1 Half-Fit

Half-Fit was proposed by Ogasawara [62] and separates the buffer classes into sizes which are a power of two. The algorithm uses a bitmap to track which Free-List has free buffers available and which does not. An example of this is depicted in Figure 6.6. The section from bit 4 to bit 10 is taken out and the corresponding Free-Lists are depicted. For instance, bit 4 is set. Therefore, there are buffers linked inside the corresponding Free-List. In contrast, bit 5 is cleared. Therefore, the corresponding Free-List is empty.

The Half-Fit algorithm assumes all buffer classes are powers of two, allowing easy calculation of the buffer class, as one has to find the highest bit set in the requested buffer size. Many processors have instructions for that operation, making it possible to have constant worst case complexity when allocating a buffer. In general however,

this cannot be assumed. Therefore, in the worst case one has to iterate over all k buffer classes to find a free buffer. Nonetheless, optimisations exist to count leading zeros with a constant worst case complexity. Freeing a buffer requires the determination of the class it belongs to and adding it to a queue. Both steps have a worst case complexity of $O(1)$. Therefore, allocating and freeing a buffer have a worst case complexity of $O(1)$.

We are now taking a look at the space required for managing the data structures behind the Half-Fit algorithm. As the Free-Lists are linked lists, their size grows linearly by the number of elements stored. Therefore, the worst case is having a fragmented memory, where all buffers are of minimum size $B_{\min}$ and the state is alternating between allocated and free. This yields a maximum buffer count $n = \frac{M}{2*B_{\min}}$. Therefore, the space requirements grow with the size of the managed memory $O(M)$.

Osagaware stated in the Half-Fit algorithm presentation that the problem of *incomplete memory usage* exists. Blocks which exceed the base size of their corresponding Free-List are not used for requests that are one byte larger than the base size of the Free-List. This is due to the handling of the separation into the different classes for the Free-Lists. The Free-List with index i has members in the size range $[2^i, 2^{i+1} - 1]$, which is used to serve request in the range $[2^{i-1} + 1, 2^i]$. Given the Free-List with index $i = 7$ holds buffers of $[2^7, 2^8 - 1]$, $2^7 + 1$ is out of its range as it is only serving $[2^6 + 1, 2^7]$. Assuming a request was issued for $2^7 + 1$ bytes, it would fail as it requires a buffer of the class $[2^7 + 1, 2^8]$, even if a buffer of size $2^8 - 1$ would be available in the smaller class. In the worst case this can yield a fragmentation of up to 50%.

6.3.5.2 Two-Level Segregated Fit

A more sophisticated member of this type of algorithms is the Two-Level Segregated Fit (TLSF) introduced by Masmano et al. [63], where the array of Free-Lists is organized in two levels. This structure is depicted in Figure 6.7. The first level provides k classes which are all powers of two. The second level then subdivides these classes linearly into additional l sub-classes. The state of the second level is encoded in a bitmap and provided to the first level to determine if the second level has free buffers available. The classes and sub-classes do not relate to distinct sizes for the buffers but size ranges, e.g., a class manages all buffers in the size range of $[2^4, 2^5 - 1]$, its sub-class holds the Free-Lists for the ranges $[16, 19]$ and $[20, 23]$.

In the following paragraph, we will take a closer look at the TLSF algorithm. First, we will assume that no splitting and merging are done. Therefore, the worst case for allocating a buffer is when a buffer of the smallest size is required, but all first level Free-Lists are empty except for the largest class. Therefore, the algorithm needs to first calculate the First Level Index (FLI) for the requested buffer size by calculating the highest set bit. This can be done by counting the leading zeros, which can be executed in constant time. One can optimise the second level buffer sizes for calculation of the Second Level Index (SLI) by using a subdivision which is a power of two, e.g., splitting each first level class into 2, 4, 8, ... second level classes. Calculation of the SLI can then

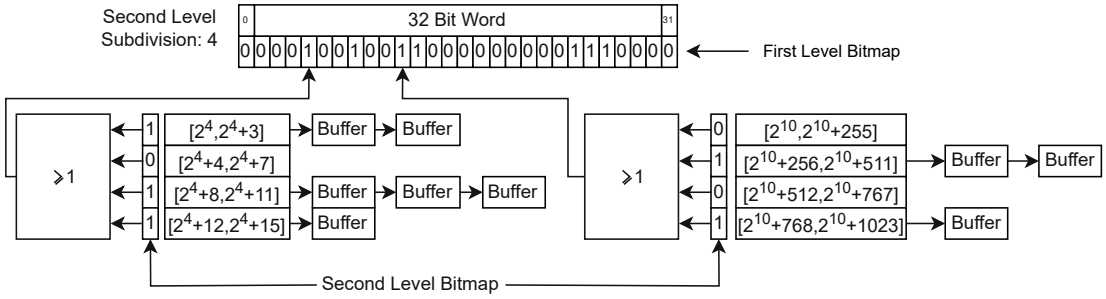


Figure 6.7: Segregated Free-Lists of TLSF, adapted from [63]

be as simple as masking out a set of bits from the size to get the index, which again can be executed in constant time. The same procedure can be applied when freeing a buffer again. Acquiring a buffer with the FLI and SLI calculated is now as simple as removing the first element from a linked list. Freeing a buffer is as trivial as it only requires appending an entry at the end of linked list. Both operations can be executed in constant time. Therefore, the algorithm has a worst case complexity of $O(1)$.

Extending the algorithm with splitting and coalescing, we need to calculate the FLI and SLI for the desired buffer size, which can be accomplished in constant time. Given that no buffer with a matching size is available, the FLI and SLI of the next bigger free buffer need to be calculated. This can be done by looking for the lowest set bit in the Free-List bitmaps, which can be executed with a constant worst case complexity. If the larger buffer exceeds the space required of a minimal buffer, it needs to be split. The newly created buffer needs to be added to its corresponding Free-List. As already introduced, the calculation of FLI and SLI as well as adding a buffer to a Free-List can be performed in constant time. In the worst case a buffer will be merged with up to two other buffers when it is freed. Removing these two buffers from their corresponding Free-Lists can be done in constant time. Therefore, the worst case complexity is still $O(1)$.

Finally, we are taking a look at the worst case space consumption for TLSF. It has a comparable worst case to the other presented algorithms, where the memory is fragmented into buffers of minimal size, which alternate between free and allocated state. Therefore, in the worst case, TLSF has a space consumption of $n = \frac{M}{2*B_{\min}}$ and the size of the linked list grows with $O(\frac{M}{2*B_{\min}})$.

As already introduced in Section 6.3.5.1, Half-Fit suffers from an inefficient classification scheme of the buffer sizes. Therefore, TLSF introduced a second level of Free-Lists, which subdivides the first layer again. Based on this, TLSF has a lower fragmentation overhead [64] and does not suffer from *incomplete memory usage*.

Algorithm Type	Uniform Buffer Sizes	Fixed Size Classes	Flexible Sized Classes	Freely Definable Buffer Sizes
Sequential Fits	X	X	X	X
Ordered binary tree best-fit		X	X	X
Buddy Systems			X	
Half-Fit		X	X	X
Two-Level Segregated Fit		X	X	X

Table 6.1: Mapping between DSA approach and type

6.4 Comparison of Memory Management Algorithms

In this section, we present a mapping between manual memory management approaches and the types of algorithms as well as a comparison between the different algorithms regarding worst case complexity and space requirements. In Table 6.1, a mapping is presented, showing how each of the algorithms introduced in Section 6.3 can be implemented with the approaches presented in Section 6.2.

The first type are *Sequential Fits*, which are rather trivial in comparison to the others and therefore, have the least requirements. In general, such an algorithm can be implemented with every DSA approach. The reason for this is, that an algorithm of this type is simply iterating over the memory and search for a matching buffer. The next type is the *ordered binary-tree best-fit* which uses a tree as its base data structure. Therefore, the approach with uniform buffers is not reasonable as the binary tree data structure would consist of a single node with a Free-List. The *binary buddy system* uses also a binary tree as its base data structure, but how the buffer classes are split is pre-defined, e.g., in powers of two or the Fibonacci sequence. Therefore, having a uniform or fixed classes has no benefit for this type of algorithm. Additionally, freely definable buffer sizes are not implementable, as how the memory is separated is pre-defined. *Half-Fit* and *TLSF* were designed to operate with freely definable memory sizes, but use classes to speed up the location of a Free-List. Therefore, both types can also be used with fixed and flexible buffer sizes, as their allocation strategy is quite efficient. Only a uniform buffer class is not suitable, as these two algorithms add unnecessary overhead with their sophisticated class calculation.

Table 6.2 shows an overview how the different algorithms compare in terms of worst case complexity and space allocation. One can see that algorithms of the *sequential fits* type, as well as the *ordered binary tree best-fit* and the *binary buddy system*, have a worst case complexity growing with the size of the managed memory M. Depending on the deployment of these algorithms, M can be considered constant, as the size of the physical memory does not change during runtime. Nonetheless, when managing memory space for a thread, it may be required to allocate further memory from the OS which changes the size of M. In contrast, the *Half-Fit* algorithm and *TLSF* have constant worst case complexity independently of the size of the managed memory.

Algorithm	Worst Case Complexity Allocation	Deallocation	Worst Case Space Allocation
Sequential Fits	$O(\frac{M}{2*B_{\min}})$	$O(1)$	$O(\frac{M}{2*B_{\min}})$
Ordered binary tree best-fit	$O(\log_2(\sqrt{M}))$	$O(\log_2(\sqrt{M}))$	$O(\frac{M}{2*B_{\min}})$
Binary Buddy System	$O(\log_2(\frac{M}{B_{\min}}))$	$O(\log_2(\frac{M}{B_{\min}}))$	$O(\frac{M}{B_{\min}})$
Half-Fit	$O(1)$	$O(1)$	$O(\frac{M}{2*B_{\min}})$
Two-Level Segregated Fit	$O(1)$	$O(1)$	$O(\frac{M}{2*B_{\min}})$

Table 6.2: Comparison between different memory management algorithms

As already explained in Section 6.3.5.1, the *Half-Fit* algorithm suffers from incomplete usage of memory. *TLSF* drastically reduces this problem with its two-level approach for the Free-Lists and rounding to the next Free-List size. In comparison, *Half-Fit* has non-external memory fragmentation of up to 50 %, whereas *TLSF* has around 3 % [64]. Fragmentation could be further reduced by sorting the entries of a Free-List by their memory address, but then the constant worst case complexity no longer holds. For instance, a sorted list has a worst case complexity of $O(n)$ for insertion, and a red-black tree has $O(\log_2(n))$. Based on this evaluation, we decided to use TLSF for the further implementation in our buffer management.

6.5 Buffer Management Design and Implementation

This section discusses the design and implementation of our manual memory management system. As our use case is related to managing buffers inside a network stack, from now on, this is referred as *Buffer Management* instead of (manual) memory management. We focus on configuration parameters and used data structures in the following section. We also describe how we optimised our version of the TLSF algorithm and how it operates.

6.5.1 Configuration Parameters

The parameter `fl_lowest_cls` defines the smallest memory unit that shall be managed. This value is interpreted as an exponent to calculate a power of two, e.g., `fl_lowest_cls=5` represents 2^5. Additionally, the parameter `fl_cls_cnt` defines how many different first level classes shall be used. The configuration value `sl_cls_cnt` defines the number of second level classes for each first level class. Again this value is interpreted to be an exponent of a power of two. Additionally, the buffer management requires configuration options which define where the memory that shall be managed is located. Therefore, the parameter `start_address` defines where managed memory starts and the parameter `end_address` defines the last usable address. In Listing 6.1, the full configuration structure is depicted. Additionally, this structure holds the Free-Lists and the availability masks required for the inner workings of TLSF. Further details can be found in Section 6.5.2.

Listing 6.1: Runtime configuration structure

```
typedef struct {
        unsigned char fl_lowest_cls;
        unsigned char fl_cls_cnt;
        unsigned char sl_cls_cnt;
        unsigned short fl_free;
        void *start_address;
        void *end_address;
        unsigned short *sl_free;
        block_header_free_tlsf_t **free_list_first;
        block_header_free_tlsf_t **free_list_last;
} tlsf_runtime_config_t;
```

6.5.2 Management Data Structures

The algorithms presented in Section 6.3 do not factor in the required overhead for managing the memory. We assume the space needed for management is located inside the memory, which is under the control of the buffer management. Considerations for the space requirements of these management structures are not negligible, as an inefficient architecture can drastically decrease the useable memory space. Therefore, when defining these structures, one must factor in the managed space's size and the minimum buffer size. The algorithms presented in Section 6.3 require a dynamic data structure, which provides the Free-Lists. The memory space required to store the members of these data structures has to be reserved when the buffer management initialises the memory.

The Free-Lists are typically implemented as linked lists. In general, an entry of a linked list consists of at least a reference to the next element, to the previous element or to both elements. Additionally, it has a reference to the data which is managed by that linked list. Without a dynamic memory management system, one has to reserve space for up to the desired maximum number of entries of said list. Given the implementation of a DSA itself, one can integrate the pointers required for operation into the Boundary-Tag of a free buffer. The additional space required does not affect the memory overhead, as it is not needed to manage an allocated buffer. Nonetheless, it requires a minimum buffer size of at least the size required for the pointers.

Additionally, the system has to track which memory areas are already allocated and which are not. During the design of the buffer management component, we came across two commonly used methods for organising these management structures. The first uses a dedicated array for tracking which fragments of the managed memory are allocated, the second uses so-called Boundary-Tags. Regarding the former variant, the managed memory space is separated into fragments of the minimum buffer size, where each buffer is a group of fragments. For each fragment, an entry in a management structure is required that tracks the allocation state and a reference to the buffer that owns this fragment. The benefit of this method is that the full space required for managing a memory space can be reserved when the mechanism is initialised. The downsides of this approach are that the fragmentation structure grows with the size of the managed

Listing 6.2: Original block header of TLSF by Masmano et al. [63]

```
typedef struct block_header{
    uint32_t size       :   30;
    uint32_t alloc      :    1;
    uint32_t last_block :    1;
    blocker_header_t *prev_phy;
} block_header_t;

typedef struct bh_free{
    block_header_t      bh;
    blocker_header_t *prev;
    blocker_header_t *next;
} block_header_free_t;
```

Listing 6.3: Size-optimised block header for the Ethernet MAC use case

```
typedef struct block_header{
    uint16_t reserved   :   2;
    uint16_t size_prev  :  14;
    uint16_t alloc      :   1;
    uint16_t last_block :   1;
    uint16_t size       :  14;
} block_header_t;

typedef struct bh_free{
    block_header_t      bh;
    blocker_header_t *prev;
    blocker_header_t *next;
} block_header_free_t;
```

memory and the time required for marking the fragments as allocated negatively impacts the buffer management's responsiveness. Additionally, this limits the sizes of the buffers, as they always have to be a multiple of the minimum buffer size.

The second method, using Boundary-Tags introduced in Section 6.1, has the benefit that the structures solely grow with the desired maximum number of buffers that shall be managed. Such a Boundary-Tag is added in before or after a buffer entry, indicating the allocation state and its size. Alternatively, the tag can be a reference where the data is stored. The usage of Boundary-Tags makes the selection of buffer classes a bit more complex, as one has to consider the additional space required for these tags. Nonetheless, this is in general a better method as it requires less space in contrast to the fragmentation structure. This mechanism can yield a better response time depending on the used fragment size.

The goal of the buffer management was to be applicable on a wide range of use cases as well as for our usage inside a network stack. Therefore, we build up on the presented data structures TLSF introduced by Nasmano et al. [63]. In Listing 6.2, a block header is depicted which is used to track memory blocks. The structure `block_header_t` consists of a `size` member to read out the size of this block, an `alloc` flag to indicate the allocation status of the buffer and a `last_block` flag to indicate if it is the last physical block. Additionally, for tracking a free block `block_header_free_t` has pointers to the previous and the next element of the corresponding Free-List. The `block_header_free_t` structure requires 16 bytes of memory, but as the pointers are only used for tracking free blocks, the management overhead is only the size of the `block_header_t` structure, which requires only 8 bytes.

As the address space of the Ethernet MAC is limited to 16 bits, we decided to design an optimised form of the block header. The memory addresses of our system are 32 bit word-aligned addresses, which means that load and store operations typically access 4 bytes at a time. We want to have the memory addresses aligned with this memory word barrier for more efficient memory access. Therefore, the lowest two bits of an address

Listing 6.4: Entries to read out availability of free buffer entries

```
uint16_t fl_free;
uint16_t sl_free[fl_cnt];
```

are always cleared, and we can omit these in our management structure without losing information. Additionally, as already introduced, our address space is limited to 16 bit; therefore, we require only 14 bit to model all possible buffer sizes. As a result, we defined the structure depicted in Listing 6.3, which generates only 4 bytes of overhead for a used block.

6.5.3 Algorithm Data Structures

For managing allocated and free buffers, we use the data structures already defined in Section 6.5.2. To manage the buffers, we need to define the data structure required by TLSF. As our `block_header_free_t` data structure provides references to the next and previous entries of the containing Free-List, we require only a reference to the list's first and last buffer entry. Therefore, our `free_list` data structure has two simple arrays holding the indices to the corresponding buffer entries. The required size for the `free_list` is the number of first level classes multiplied by the number of second level classes.

To optimise the process of acquiring new buffers, bitmaps are used to indicate the state of each level. The complete memory space of the Ethernet MAC is 57344 bytes. Therefore, a first level class for representing $[2^{15}, 2^{16} - 1]$ is required by the definition of TLSF. The bitmap representing the first level requires a size of 16 bit. To give the user more freedom when configuring the second level, we decided to allow up to 16 second level classes. Therefore, the maximum number of first and second level classes is limited to 16 entries which fit into an 8 bit data type. If the bit corresponding to the buffer class is set in the bitmap, a free buffer entry to be acquired exists. The bits marking availability inside the mask correspond to the FLI and SLI. The data types used are depicted in Listing 6.4.

6.5.4 Memory Overhead

In Table 6.3, we calculated the memory space required to fit our algorithm and management structures. The table depicts which member of what structure requires how much space and how much space the entire structure in the required amount takes. As the base of our calculation, we used the following configuration, see Section 6.5.1: `fl_lowest_cls=5`, `fl_cls_cnt=11`, `sl_cls_cnt=2`. We chose `fl_lowest_cls=5`, as the smallest frame the network stack is able to send is an ARP message, which requires 34 bytes. To have a buffer with the remaining size of the Ethernet MACs memory, we require the full 16 bit address space and, therefore, `fl_cls_cnt=11`. Finally, for the subdivision of the second level we decided to use `sl_cls_cnt=2`. This has the benefit

Structure	Member	Size (bytes)	Count	Total (bytes)
Runtime				28
Configuration	`fl_lowest_cls`	1	1	
	`fl_cls_cnt`	1	1	
	`sl_cls_cnt`	1	1	
	Padding	1	1	
	`fl_free`	2	1	
	Padding	2	1	
	`start_address`	4	1	
	`end_address`	4	1	
	`sl_free` address	4	1	
	`free_list_first` address	4	1	
	`free_list_last` address	4	1	
Bitmaps				88
	`sl_free`	2	44	
Free				352
Lists	`free_list_first`	4	44	
	`free_list_last`	4	44	
`block_header_t`		4	255	1020
	`reserved`	2 bit	1	
	`size`	14 bit	2	
	`alloc`	1 bit	1	
	`last_block`	1 bit	1	
Total space required				1488

Table 6.3: Memory space consumption of TLSF, configuration: `fl_lowest_cls=5`, `fl_cls_cnt=11`, `sl_cls_cnt=2`, assuming 255 buffers are in use

of producing less overhead when dealing with smaller buffers. In the last row of Table 6.3, one can see that the total memory space required is 1488 bytes. In our environment with the Ethernet MAC, we have 57344 bytes of memory available. Therefore, our management structures use about 2.59% of the overall managed memory space.

6.5.5 Calculation of First and Second Level Index

A crucial step of TLSF is to calculate the First Level Index (FLI) and Second Level Index (SLI) to which the buffer size belongs. As the classes of the first level are all powers of two, one has to find the highest bit set in the given size value. This can be done by calculating the $\log_2$, but this requires using floating point operations. As our target does not have a Floating Point Unit (FPU), executing floating point operations is quite inefficient. Most modern CPUs have instructions to find the highest bit set.

Listing 6.5: Calculate the highest bit set, Henry Warren's "Hackers Delight" [65]

```
uint32_t nlz(uint32_t x) {
    uint32_t n = 32;
    if ((x>>16) != 0) { n=n-16; x = x>>16; }
    if ((x>>8 ) != 0) { n=n- 8; x = x>> 8; }
    if ((x>>4 ) != 0) { n=n- 4; x = x>> 4; }
    if ((x>>2 ) != 0) { n=n- 2; x = x>> 2; }
    if ((x>>1 ) != 0) { return n-2; }
    return n - x;
}
```

Unfortunately, this is also not supported by our target CPU. Therefore, we decided to adapt a method presented by Henry Warren in the Book "Hackers Delight" [65], seen in Listing 6.5. The return value of the `nlz` function is the count of leading zeros of a binary number. Subtracting this value from the size of the underlying data type yields the position of the highest set bit. For instance, if we call `nlz` with 15 as input, the function returns 28 since the binary representation of 15 is `1111` and the used integer is 32 bits wide. Subtracting 28 from 32 yields 4, i.e., the 4th bit is the highest set bit. With that, we can easily calculate the FLI. We selected this approach as it is more efficient than iterating over the number and checking for the highest bit set.

In order to improve the access time of our TLSF implementation, the algorithm was changed to an algorithm presented by EmbeddedGurus[3] depicted in Listing 6.6. It uses a lookup table, which is used to simplify the five if-conditions from Listing 6.5 to three if-conditions, where two will be taken at maximum. It searches for the highest byte where a bit is set and the value of said byte is extracted, which will then be used as an index for the lookup table. Reducing the value from the table by the shift amount provides the number of leading zeros. The original version requires 930 cycles the improved version only requires 312 cycles.

For easier calculation of the SLI, we decided to limit the number of second level classes to be a power of two. Listing 6.7 shows how the value can be calculated using arithmetic and bitwise operations. First, the size value is shifted to represent the SLI value inside the least significant bits. The shift amount is the FLI decremented by the bits required for the SLI. Afterwards, all bits except the ones for the SLI are cleared. This approach has the benefit that it does not require any branching or looping to determine the SLI.

If there is no free buffer for the determined FLI and SLI, new indices have to be calculated. There are two possible cases: either a larger buffer exists in the Free-List for the calculated FLI, or no buffers are left for this FLI. In the former case, we have to adapt the SLI to find the next larger second level Free-List of the calculated FLI that is not empty. The process is depicted in Listing 6.8. Therefore, we create a mask that clears all second level buffer entries with a lower SLI from the availability mask. In the next step, we determine

[3] `https://embeddedgurus.com/state-space/2014/09/fast-deterministic-and-portable-counting-leading-zeros/` last accessed 11.07.2022

Listing 6.6: EmbeddedGurus optimised implementation of calculating the highest bit set

```
uint32_t nlz(uint32_t x) {
    static uint8_t const clz_lkup[] = {
        32U, 31U, 30U, 30U, 29U, 29U, 29U, 29U, ...
    };
    uint32_t n;
    if (x >= (1U << 16)) {
        if (x >= (1U << 24)) {
            n = 24U;
        }
        else {
            n = 16U;
        }
    }
    else {
        if (x >= (1U << 8)) {
            n = 8U;
        }
        else {
            n = 0U;
        }
    }
    return (uint32_t)clz_lkup[x >> n] - n;
}
```

Listing 6.7: Calculate the Second Level Index (SLI) from initial FLI

```
fli = 16 - nlz(size);
uint16_t shift_amount = (fli - sl_cls_cnt);
uint16_t tmp = size >> shift_amount;
sli = tmp & ((1 << sl_cls_cnt));
fli -= fl_lowest_cls;
```

Listing 6.8: Calculate the adapted SLI

```
uint16_t tmp = sl_free[fli] & ~((1 << (sli+1)) - 1);
tmp &= (~tmp)+1;
sli = 16 - nlz(tmp)-1;
```

the two's complement and use it to mask away all set bits except the lowest set. From this Free-List, we shall take our buffer. As for calculating the FLI, we can use the `nlz` function to get the number of leading zeros and calculate the set bit position.

As already introduced, no buffer may be available inside the calculated FLI. Therefore, a new FLI and new SLI has to be calculated. To get the new FLI, we adapted the procedure presented in Listing 6.8. Therefore, we first mask away all bits lower than our calculated FLI. Afterwards, the two's complement is determined to mask away every bit except for the lowest set. To get the new SLI, we use the same approach with the two's complement to get the lowest set bit. The adapted process is depicted in Listing 6.9.

Listing 6.9: Calculate the adapted FLI and SLI

```
uint16_t tmp_fli = fl_free & ~((1 << (fli+1)) - 1);
tmp_fli &= (~tmp_fli)+1;
fli = 16 - nlz(tmp_fli)-1;
uint16_t tmp_sli = config->sl_free[fli] & ((~config->sl_free[fli]) +1);
sli = 16 - nlz(tmp_sli)-1;
```

Listing 6.10: Calculate the adapted FLI and SLI

```
void * tlsf_malloc(size){
        fli, sli = get_next_free_fli_sli(size);
        buffer = remove_buffer_from_free_list(fli, sli);
        if(buffer->size > size + min_siz)
        {
                new_buffer = split_buffer(buffer, size)
                add_buffer_to_free_list(new_buffer);
        }
        return buffer;
}
```

6.5.6 Getting a Buffer

With the FLI and SLI calculated as described in Section 6.5.5, we have calculated the required indices to access the data structures introduced in Section 6.5.2 and Section 6.5.3. In Listing 6.10, a simplified code version of the following steps is depicted. If the calculated FLI and SLI point to a buffer larger than the required size, the buffer may be split. Therefore, the buffer has to be at least the required size plus the minimum buffer size and the space required for the Boundary-Tag. Additionally, the newly created buffer needs to be added to its corresponding Free-List.

6.5.7 Freeing a Buffer

A simplified coded version of the following steps is depicted in Listing 6.11. When freeing a buffer, the relative address of the buffer to be freed and its physically next and previous buffer have to be calculated. If any of these buffers are not allocated, a merge of buffers will happen, which may merge up to three buffers. The merged buffer is then added to its corresponding Free-List.

6.6 Further Findings and Reflections

As this thesis has a certain scope, not all cases applicable to a memory management algorithm can be handled. This section discusses the limitations and delimitations of our design and implementation. Additionally, we describe some pitfalls regarding the used hardware and its software side.

Listing 6.11: Freeing a buffer and merging it if neighbours are not allocated

```
void tlsf_free(buffer){
        next_buffer = buffer + buffer->size;
        prev_buffer = buffer - buffer->prev_size;
        if(next_buffer->alloc = 0)
        {
                remove_buffer_from_free_list(next_buffer);
                merge_buffers(buffer, next_buffer)
        }
        if(prev_buffer->alloc = 0)
        {
                remove_buffer_from_free_list(prev_buffer);
                merge_buffer(prev_buffer, buffer);
                buffer = prev_buffer;
        }
        add_buffer_to_free_list(buffer);
}
```

6.6.1 Usage inside 64-Bit Systems

The presented solution can generally be used for 64-bit systems without adaptions, but some points need to be considered beforehand. Given that the system has a 64-bit word boundary, the presented structures in Section 6.5.2 are suboptimal. The `block_header_t` structure still requires 4 bytes in size, but when it is used inside the `block_header_free_t`, padding is inserted by the compiler. This is done, as the pointers `prev` and `next` require 8 bytes, and the compiler tries to have optimal memory access. Having the members aligned with the word boundaries only requires a single memory access instead of two if the member crosses word boundaries. With the added padding and the twice-as-long pointer address, the structure now requires 24 bytes in memory instead of 12 bytes. Due to the increased memory size, the minimal buffer space has to be set to 32 bytes as otherwise the structure `block_header_free_t` would not fit inside the free buffer. Another way to solve this issue is to adapt the block header to 8 bytes instead of 4. Then the smallest possible buffer can still be 16 bytes, but this would double the memory overhead for the block header.

Additionally to the larger required buffer, the memory access generally is worse than with the 32-bit system. As already introduced, the `block_header_t` requires only 4 byte, which is less than a word. The implemented TLSF algorithm now returns a buffer address, which is aligned to a half word and not to full word. This can lead to a drop in memory bandwidth if the compiler expects a word align memory address from the buffer management. In general, this is less of a problem for our use case, as this is mainly relevant when working with data which uses 64-bit numbers.

The calculations presented in Section 6.5.4 focus on a 32-bit system and are not applicable to the 64-bit system. Assuming a block header is used to get optimal memory access and the pointers take twice the space, the required memory space for storing the data structures nearly doubles.

6.6.2 Security Concerns

Due to the RT nature of the use case, no clearing of the memory space has been implemented, i.e., when a buffer is reused, the old data is still in the buffer. This is not an issue for our dedicated use case in a network stack, as no buffer is provided to an application. If this memory management implementation is used to manage main memory, this could lead to leaking encryption and decryption keys. As clearing a buffer depends on its size and cannot be done in constant execution time, this was not implemented.

6.6.3 Delimitations Regarding Multicore and Multithreaded Systems

All the buffer management systems from Section 6.3 rely on data structures to ease the location of free memory space as well as splitting and merging buffers. In multicore and multithreaded environments, these data structures must be accessible to multiple consumers and producers. An RTS requires synchronisation mechanisms without locks, starvation and deadlocks. Such an object or algorithm is called *wait-free,* which is the strongest non-blocking guarantee regarding an actor's progress. Such structures require atomic access to the memory, typically handled with hardware instructions like Compare-and-Swap (CAS) or Load-Link/Store-Conditional (LL/SC). These instructions check whether the desired memory area contains the expected value and modify it, only if another actor did not modify the expected value. Unfortunately, the underlying hardware platform Patmos [20] does not support the required atomic instructions. Therefore, an implementation of a such a wait-free queue is not feasible.

6.6.4 Patmos Address Spaces

The CPU architecture of Patmos uses different typed store and load functions depending on the location of the memory. Into which memory space a pointer points must be communicated to the compiler when the program is compiled. By default, a pointer always points to the main memory, but an IO device is located in a different address space requiring a different load/store instruction. Therefore, the used pointers inside the TLSF implementation need an additional attribute `_IODEV`. Otherwise the access does not work. To use this implementation also for main memory, the code has to be copied and said attribute must be removed.

6.7 Evaluation

In this section, we elaborate how we evaluate our proposed buffer management. Therefore, we take a look into two aspects: first, how it performs in a static Worst-Case Execution Time (WCET) analysis, second, we measure the average time to allocate a buffer.

Function	WCET n cycles	WCET at 80 MHz
`buffer_management_malloc ()`	19609	245.11 µs
`tlsf_malloc ()`	18850	235.62 µs
`get_buffer ()`	9473	118.41 µs
`get_next_free_fli_sli ()`	3625	45.31 µs
`remove_buffer_from_free_list ()`	3874	48.43 µs
`split_buffer ()`	3219	40.24 µs
`add_buffer_to_free_list ()`	3659	45.74 µs
`get_fli_sli ()`	1547	19.34 µs
`buffer_management_free ()`	20734	259.18 µs
`tlsf_free ()`	20151	251.86 µs
`merge_buffers ()`	17662	220.78 µs
`remove_buffer_from_free_list ()`	3874	48.43 µs
`add_buffer_to_free_list ()`	3659	45.74 µs
`add_buffer_to_free_list ()`	3659	45.74 µs
`get_fli_sli ()`	1547	19.34 µs

Table 6.4: WCET analysis values of main functions and their major sub-functions

6.7.1 Static WCET Analysis

The main task of the buffer management is taking care of the allocation and deallocation of buffers. Therefore, we take a look at `tlsf_malloc` to determine how long the allocation of a buffer can take at maximum. Finally, we will also look at `tlsf_free` as it takes care of deallocating and merging buffers. The results of the WCET analysis for these functions and their major sub-functions are listed in Table 6.4.

The static WCET analysis reveals that, at worst, the buffer allocation takes 232.3 µs. This includes the actual allocation of a buffer (`get_buffer`) by calculating the next best class, which has a buffer large enough to satisfy the request (`get_next_free_fli_sli`) and removing the buffer from the Free-List (`remove_buffer_from_free_list`). Additionally, this may require the buffer to be split (`split_buffer`) if the overhead is larger than the space required for a minimal buffer. This newly created buffer has to be added to the corresponding Free-List (`add_buffer_to_free_list`), which requires the calculation of the buffer class affiliation (`get_fli_sli`).

If we want to free an allocated buffer again, this takes up to 251.86 µs in the worst case. The largest contributor to this worst case is the possibility of being required to merge up to three free buffers, which is executed by `merge_buffers`. The main components of these functions are removing buffers that shall be merged from their corresponding Free-Lists (`remove_buffer_from_free_list`) and adding the newly created buffer to its corresponding Free-List.

6.7.2 Average Runtime

A worst case complexity of $O(1)$ does not imply a constant runtime. Therefore, we investigate how long allocation and deallocation of buffers take on average. We have written a test program that measures how long these functions take for different buffer sizes. We decided to probe the following scenarios:

1. a buffer already exists for each class
2. an empty memory pool
3. another buffer already exists

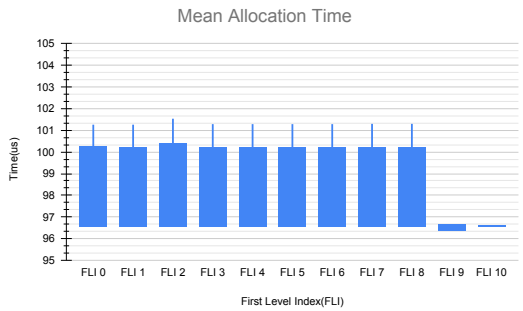
In the first case, we evaluate how efficient the process of allocating and freeing buffers is if no splitting or merging is required. In the second case, we can evaluate if all buffer classes take the same effort to split. In the third case, we can track how merging with up to two buffers instead of one can affect the runtime.

For the measurements, we used the same configuration values as shown in Table 6.3: `fl_lowest_cls=5`, `fl_cls_cnt=11`, `sl_cls_cnt=2`. Therefore, we have a total of 44 buffer classes to check for these scenarios. To improve the readability of the charts, we group the four second level classes of each first level class. This reduces the charts from 44 columns to 11, i.e., the horizontal axis depicts the First Level Index (FLI) from zero to ten. Since the memory space of our test environment is not large enough to test all classes, the largest two classes could not be tested. For the box plots seen in Figure 6.8, 100 samples were taken for each class an each test case.

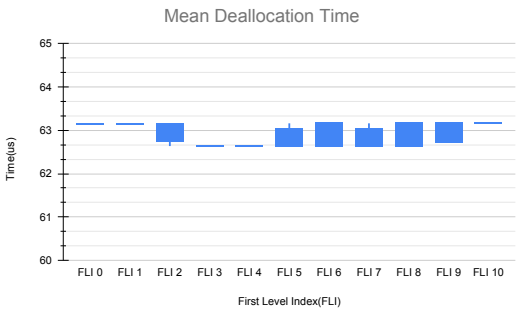
In Figure 6.8, the measurements of the three defined scenarios are depicted. Figure 6.8a and Figure 6.8c depict the measurements for allocations and Figure 6.8b and Figure 6.8d the corresponding measurements for the deallocations.

In Figure 6.8a, we can see the mean time to allocate a buffer. The individual second level classes are grouped and named as the corresponding first level class index, i.e., the FLI. We can see that an allocation where no split of a buffer is required, takes between 96.5 µs and 101.5 µs. When taking a look at the times from the single classes, we can see that the maximum values come from the largest second level classes. When a buffer shall be allocated, a buffer from the next larger class is taken as it is guaranteed that the buffer is large enough for the requested size. If we allocate a buffer of the SLI 3 it has to check the next FLI for a buffer. In Figure 6.8b, the mean time to free a buffer that does not get merged with a neighbouring buffer is depicted. We can see that this takes between 62.5 µs and 63.5 µs. The minor fluctuations can be backtracked to the different paths taken for the calculation of the FLI and SLI.

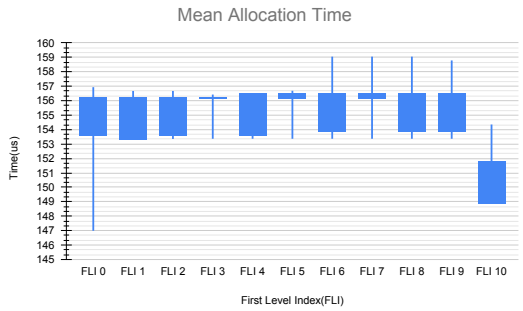
In Figure 6.8c, we have depicted the mean time to allocate a buffer, if the managed memory pool is empty. In other words, each allocation needs to split the buffer into two buffers. We can see that the mean allocation time is between 153 µs and 158 µs. The



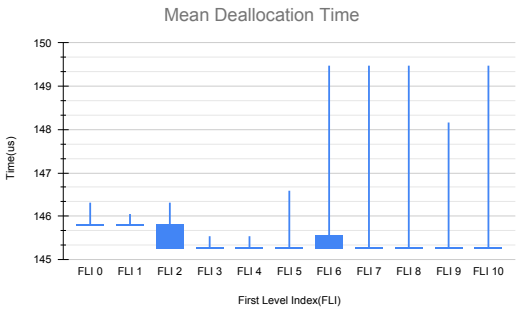
(a) Mean allocation time without splitting a buffer



(b) Mean deallocation without merging buffers



(c) Mean allocation time with splitting a buffer



(d) Mean deallocation with merging buffers

Figure 6.8: Allocation and deallocation measurements for various buffer sizes and scenarios

outliers of *FLI 0* and the runtime of the whole class *FLI 10* shows some unexpected behaviour which requires further investigation.

In Figure 6.8d, the mean time to free buffer and merge it with its neighbouring buffers is depicted. As the scenario always forces three buffers to merge, the runtime is fairly steady between 145 µs and 146 µs. The classes starting with *FLI 6* have significantly larger maximums, in relation to the classes until *FLI 5*. Some further investigation into the behaviour would be required to find the root cause of the spikes.

6.7.3 Runtime over Time

Additionally to the isolated measurements, we tracked how allocations and deallocations perform over longer periods. A test program decides at random if a buffer shall be allocated or freed. If the program shall allocate a buffer, a random buffer size is used and added to an array storing all buffers allocated by the test program. If a buffer shall be freed, a random entry from the prior mentioned array is taken and handed over to the buffer management to be freed. For the measurement, we additionally track the amount of used memory.

Figure 6.9a depicts how long a buffer allocation takes distributed over the current

(a) Allocation time distributed over the currently allocated memory

(b) Deallocation time distributed over the currently allocated memory

allocation state of the managed memory. One can identify two denser populated time intervals in the chart. The spots at around 100 µs overlap with the measurements from Figure 6.8a, where we measured the mean allocation without splitting a buffer. Additionally, at around 150 µs we have overlaps with Figure 6.8c, where we have taken a look at allocations where it is required to split a buffer. The variance is a side effect of the calculations of the FLI and SLI if no free buffer is available for the originally calculated indices.

In Figure 6.9a, the distribution of the time it takes to free a buffer over the current allocation state of the memory is depicted. In the chart one can see three denser intervals, where the lowest at around 60 µs to 65 µs overlaps with the measurements depicted in Figure 6.8b, which correspond to a freed buffer that is not merged with another buffer. For the time intervals from around 130 µs to 150 µs overlap with the measurements depicted in Figure 6.8d, where each buffer is merged with two other buffers. Therefore, the denser spots starting at 110 µs up to around 120 µs show buffers which are only merged with one other buffer. At around 150 µs there are numerous outliers, which hint that this is the same underlying behaviour that was already encountered at the runtime measurements depicted in Figure 6.8d. It should be noted that this is still within the WCET which is 251.86 µs. These measurements further confirm our findings that the TLSF algorithm has a worst-case complexity of $O(1)$.

6.8 Summary

In this chapter, we introduced the subject of memory management and its main building blocks, see Section 6.1. Based on this, we provided an extended categorisation of approaches used in manual memory management in Section 6.2. Derived from the approaches, we introduced commonly found algorithm types used for Dynamic Storage

Allocation (DSA) with their benefits and drawbacks, as well as rough estimations for their worst case complexity, see Section 6.3. In Section 6.4, we compared the various algorithm types to each other and examined how they fit into the categories from Section 6.2. Based on the comparison, we selected Two-Level Segregated Fit (TLSF) as the algorithm we want to use for the buffer management in our software-based TSN end station. In Section 6.5, our design and implementation of TLSF is presented with its delimitations and limitations listed in Section 6.6. Our implementation was evaluated using a static WCET analysis and runtime measurement. The results can be found in Section 6.7.

CHAPTER 7

Time-Sensitive Networking

As already introduced in Section 4.2.1.1, Ethernet [2] is one of the broadest used network standards. Unfortunately, this protocol is not applicable for RT applications due to its indeterministic timing behaviour. Therefore, the standards for TSN define mechanisms like traffic scheduling to improve Ethernet for establishing determinism.

In this chapter, we provide an introduction to the basics of TSN in Section 7.1. From the standards defining TSN, we derive the central elements for the design of our TSN network interface, see Section 7.2. Based on these essential elements, we design and implement our TSN-aware driver in Section 7.3. In Section 7.4, we outline further findings during the implementation of the driver.

7.1 Introduction to TSN

TSN is a set of standards developed by the *Time-Sensitive Networking task group* of the IEEE. Audio Video Bridging (AVB) [66] provides the fundamentals IEEE uses to specify TSN. It is not a single standard, but a set of standards and amendments to already existing ones. Most specifications relevant to this thesis are amendments to IEEE 802.1Q [67], which focuses on network bridges and bridged networks.

TSN generally has three basic building blocks: time synchronisation, traffic scheduling and network configuration. Time synchronisation via gPTP [26] establishes a common notion of time. Since we already gave a detailed introduction to this topic, please refer to Chapter 5 for more information. Time Synchronisation is vital as all network members with schedules should be synchronised. This synchronised state is relevant as the schedules running on the individual devices shall be well aligned. See Section 7.1.1 for further information. The schedules shall be adaptable when a new network member joins or an application starts. Therefore, network configuration utilities are required, which get introduced in Section 7.1.2.

7.1.1 Traffic Scheduling

Ethernet's design centres around multiple devices accessing the same transmission medium, called the ether. Ethernet uses CSMA/CD [2] to handle access to the ether without guarantees regarding successful transmission. CSMA/CD senses if there is currently a frame in transmission. The sender may only start transmitting if this is not the case. Nonetheless, multiple senders could sense that there is no frame on the wire and start their transmissions simultaneously. When a sender detects a collision, the transmission will stop and restart after a random time interval. Modern Ethernet networks use switches to connect multiple devices. Therefore, only two devices have a physical connection, drastically reducing the collision probability. Modern switches have routing tables, VLAN support, and more functionalities, which add delays and do not guarantee transmission. This transmission mode is commonly known as BE.

TSN provides multiple types of traffic scheduling to improve the latency and guarantee for RT traffic. It defines three traffic types, the already introduced BE traffic, time triggered, and rate-constrained traffic [8]. To limit a stream which may transmit large amounts of data such that there is enough bandwidth left for the remaining streams, one may configure this as rate-constrained traffic. TSN provides this functionality using the credit-based shaper algorithm or the Enhanced Transmission Selection Algorithm. In order to provide RT guarantees to a stream, TSN also provides time-triggered traffic. Therefore, the network devices or a single entity creates a schedule when which traffic class can send data. These traffic classes isolate the various forms of traffic such that BE traffic cannot affect rate-constrained or time-triggered traffic.

7.1.2 Network Configuration

The term *network configuration* is quite broad and includes many different types and possibilities. Looking at the Dynamic Host Configuration Protocol (DHCP), which allows one to dynamically configure a host's IP address and distribute various information. This data includes Domain Name System (DNS) server addresses which can be used for name resolution to find services on the Internet. Another protocol for this task is the Simple Network Management Protocol (SNMP), which originated in the domain of network management and monitoring. Configuration of it is pretty cumbersome. Nowadays, its primary use is for monitoring reasons.

NETCONF [68] is a new standard to overcome the shortcomings of SNMP. For configuration, NETCONF exchanges XML-encoded messages via Remote Procedure Call (RPC) over a Secure Shell (SSH) tunnel. Using SSH adds an encryption layer to protect messages from malicious third parties on the network. Typically, YANG [69] models provide the underlying means to define what interface parameters are configurable. The TSN working group of the IEEE is constantly working on defining and improving models to configure TSN devices. Table 7.1 includes amongst other standards, the amendments regarding NETCONF and YANG models.

Short-Title	Title	Ref.
IEEE 802.1AS 2020	Timing and Synchronization for Time-Sensitive Applications	[26]
IEEE 802.1AB 2016	Station and Media Access Control Connectivity Discovery	[70]
IEEE 802.1ABcu 2021	Amendment: YANG Data Model	[71]
IEEE 802.1ABdh 2021	Amendment: Support for Multiframe Protocol Data Units	[72]
IEEE 802.1AX 2020	Link Aggregation	[73]
IEEE 802.1BA 2021	Audio Video Bridging (AVB) Systems	[66]
IEEE 802.1CB 2017	Frame Replication and Elimination for Reliability	[74]
IEEE 802.1CBcv 2021	Amendment: Information Model, YANG Data Model and Management Information Base Module	[75]
IEEE 802.1CM 2018	Time-Sensitive Networking for Fronthaul	[76]
IEEE 802.1CMde 2020	Amendment 1: Time-Sensitive Networking for Fronthaul	[77]
IEEE 802.1CS 2020	Link-local Registration Protocol	[78]
IEEE 802.1Q 2018	Local and Metropolitan Area Networks–Bridges and Bridged Networks	[67]
IEEE 802.1Qbu 2016	Amendment: Frame Preemption	[79]
IEEE 802.1Qbv 2015	Amendment: Enhancements for Scheduled Traffic	[8]
IEEE 802.1Qca 2015	Amendment: Path Control and Reservation	[15]
IEEE 802.1Qch 2017	Amendment: Cyclic Queuing and Forwarding	[16]
IEEE 802.1Qci 2017	Amendment: Per-Stream Filtering and Policing	[17]
IEEE 802.1Qcc 2018	Amendment: SRP Enhancements and Performance Improvements	[80]
IEEE 802.1Qcp 2018	Amendment: YANG Data Model	[81]
IEEE 802.1Qcr 2020	Amendment: Asynchronous Traffic Shaping	[82]
IEEE 802.1Qcx 2020	Amendment: YANG Data Model for Connectivity Fault Management	[83]

Table 7.1: List of relevant IEEE standards and amendments for TSN

7.1.3 Base Standards and Contributing Standards

As already introduced, TSN is not a single standard but a large set of standards and amendments forming it. Table 7.1 lists all standards contributing to TSN. One can see that there is a large variety of functionality defined here. Generally, the standards consist of two larger groups, the functionality behind TSN and configuring TSN networks. This thesis focuses on the IEEE 802.1AS and IEEE 802.1Qbv, as these standards define the base functionality of TSN.

7.2 TSN Network Interface Design

In the following section, we focus on the elements of a TSN-capable network interface. Typically, this is referred to as a *port* instead of an *interface*. A port commonly refers to the hardware elements like the PHY chip and the MAC. A network interface also refers to the virtual representation, like the driver and the representation inside the network

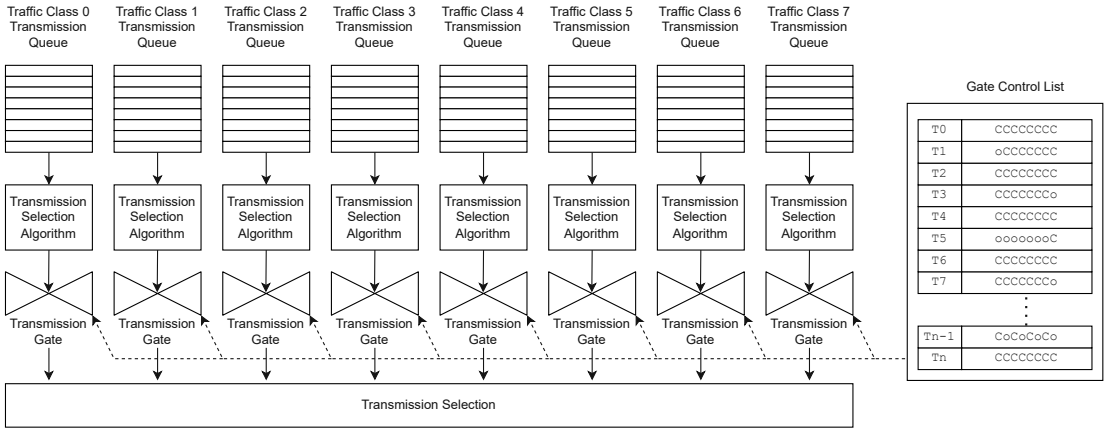


Figure 7.1: Example network interface with traffic scheduling

stack. Since this thesis focuses on a software-based implementation, we decided to use the term interface in this chapter. A TSN interface design includes the basic architecture of a network interface. In addition to the architecture, each interface gets managed by a small set of state machines.

7.2.1 Interface Architecture

To provide low latency network transmission over standard Ethernet, TSN uses traffic scheduling as introduced in Section 7.1.1. Figure 7.1 depicts an example of a network interface with the components defined by TSN for traffic scheduling. Each interface has between one and eight traffic classes. With these classes, one can isolate specific traffic or impose priorities, e.g., safety-related signals or set bandwidth limitations. Limiting the bandwidth can be helpful to prevent streams from cannibalising the link's bandwidth and provide applications with guarantees about the available bandwidth.

Each queue has a set of *Transmission Selection Algorithms (TSAs)* available. These algorithms manage whether the queue will forward frames to the transmission gate or not. Three types of TSAs defined: *strict priority algorithm*, *credit-based shaper algorithm* and *Enhanced Transmission Selection Algorithm*. The *strict priority algorithm* adds no further logic at the end of a queue and lets everything through. It simply defines that the queue with the highest priority gets sent out first, which is taken over by the traffic selection. The *credit-based shaper algorithm* and *Enhanced Transmission Selection Algorithm* manage that a queue has allowed bandwidth and transmission rate. If a queue exceeds its limits, the TSA blocks new frames from the transmission and waits for a new interval.

After the TSA, the transmission gate is the next step, which acts like a valve that is either opened or closed. This queue will send no message if the gate is closed. If the gate is open, messages may get removed from the queue. Which transmission gate shall be

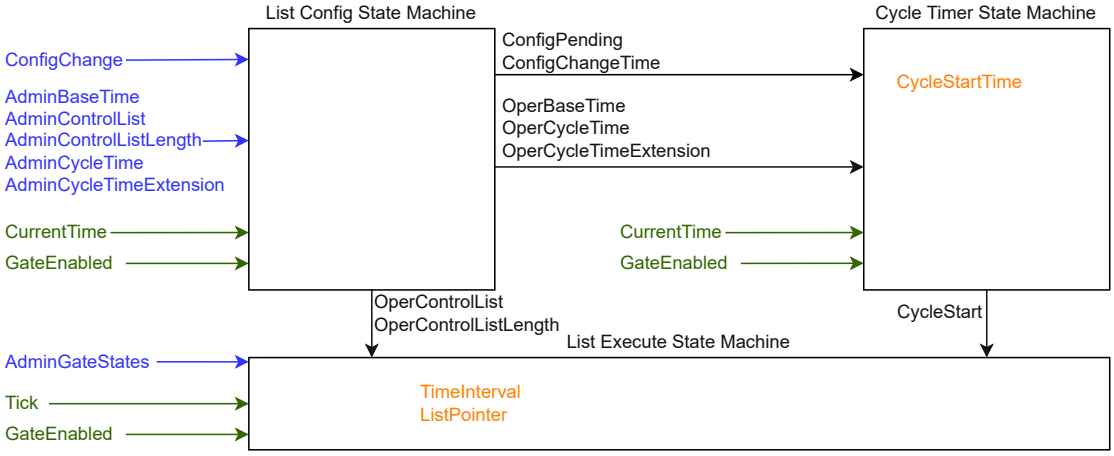


Figure 7.2: State machines of a TSN interface

opened or closed is determined by the Gate Control List (GCL) and when the state shall change. It holds each gate's state over the schedule's timespan.

When a transmission has finished, the *transmission selection* has to select which frame to send next. It selects the next frame based on the transmission gate state and which queues contain frames. From all open queues, the transmission selection takes a frame from the queue with the highest priority containing at least one frame. The priority is in ascending order. Therefore, a higher queue number yields a higher priority. For example, if queues 0 and 5 are both open, a frame will be taken from queue 5 because of its higher priority. If queue 5 does not contain a frame, the transmission selection will take a frame from queue 0, if available.

7.2.1.1 Interface State Machines

Additionally to the components in Figure 7.1, each interface shall have three state machines: the *cycle timer state machine*, the *list execute state machine* and the *list config state machine*. Figure 7.2 depicts simplified versions of these state machines and how they interact with each other. The coloured text with arrows indicates variables, blue identifies configuration input, green marks inputs for the general operation, orange is for internal variables of the state machine, and black indicates values exchanged between the state machines.

The *GateEnabled* identifier defines whether the whole process shall be active or not. The *CurrentTime* variable provides the current local time of the interface to the state machines. A derivative from that is the *Tick* variable, which is a boolean variable and gets set every nanosecond. Additionally, most variables have one of two prefixes: *Admin* or *Oper*. *Oper* specifies an operational variable, i.e., it is used during operation. The prefix *Admin* defines an administrative value and is a temporary configuration variable. When the configuration gets updated, these variables get copied to their *Oper* counterpart.

The *List Config State Machine* updates the operational values used by the other two state machines. *AdminControlList* and *AdminControlListLength* provide the new definition to update the currently used GCL. In contrast, *AdminCycleTimeExtension* provides how long the system is allowed to extend its cycle time to update the GCL.

When the *ConfigChange* input gets set, the list config state machine initiates a configuration change. Afterwards, it sets the *ConfigPending* variable to indicate that the configuration change is in progress. If the provided *AdminBaseTime* is later than the *CurrentTime*, then the *ConfigChangeTime* will be set to the *AdminBaseTime*. If not, the state machine calculates the *ConfigChangeTime* by adding the *AdminCycleTime* to the *AdminBaseTime* until it exceeds the *CurrentTime*. When *CurrentTime* reaches *ConfigChangeTime*, the *Admin* variables get copied to their *Oper* counterparts. After this step, the *ConfigPending* and *ConfigChange* variables get cleared again.

The next block is the *Cycle Timer State Machine*. Its main task is to calculate the next *CycleStart* timestamp to start the subsequent execution of the operational GCL. If no configuration change is pending, the state machine calculates the next *CycleStartTime* by adding *OperCycleTime* to *OperBaseTime* until it exceeds the *CurrentTime*. Suppose a configuration change is pending, and there is enough time left until *CurrentTime* reaches *ConfigChangeTime*. In that case, the state machine continues using the old schedule. If a configuration change is pending, and there is not enough time left until *CurrentTime* reaches *ConfigChangeTime*, then the new *CycleStartTime* is *ConfigChangeTime*. When *CurrentTime* exceeds the calculated *CycleStartTime*, the variable *CycleStart* gets assigned and triggers the *List Execute State Machine*.

The final block is the *List Execute State Machine*, which handles the execution of the GCL and controls the transmission gates. It has two internal variables, the *ListPointer* and the *TimeInterval*. The *ListPointer* is the index of the current entry of the *OperControlList*, and the *TimeInterval* variable holds the duration of the current GCL entry. If no GCL is available, the state machine uses the *AdminGateStates* configuration. When the *CyclStart* variable gets set, the state machine starts with the execution of a new cycle. It reads the *OperControlList* and *OperControlListLength* provided by *List Config State Machine* and resets the *ListPointer*. Additionally, it copies the duration of the first entry of the *OperControlList* into *TimeInterval*. Every time the *Tick* variable gets set, it decrements the *TimeInterval*. If it reaches zero, it increments the *ListPointer* and loads the next entry of the *OperControlList*. If the *ListPointer* reaches the value of *OperControlListLength*, the state machine stops further execution. Afterwards, the state machine waits until the *CyclStart* variable gets set again.

7.3 Implementation of a TSN-aware Driver

Chapter 4 focused on designing and implementing an RT network stack's protocol elements and application interfaces. In this section, we focus on the implementation of a TSN driver. The following section discusses the elements required for operation as defined in IEEE

Listing 7.1: Queue entry structure

```
typedef struct
{
  void *buffer_address;
  unsigned short padding;
  unsigned short frame_length;
} queue_entry_t;
```

Listing 7.2: Queue structure

```
typedef struct
{
  queue_entry_t *queue;
  unsigned char read_ptr;
  unsigned char write_ptr;
  unsigned short length;
} queue_t;
```

802.1Q [67]. We assume for the remaining chapter that time synchronisation is already established. For further information on time synchronisation, please see Chapter 5.

7.3.1 Traffic Queues

TSN requires separate queues for the individual traffic classes to provide traffic scheduling. Therefore, we defined the structures defined in Listing 7.1 and Listing 7.2. Each queue consists of a set of entries, which the structure `queue_entry_t` provides. It holds the address where a frame is stored and the length of the frame. We added an additional `padding` field, which reserves memory and can be used in the future if required. Nonetheless, the compiler would add an implicit padding field for better memory access. In order to prevent the cannibalisation of the whole memory of the Ethernet MAC by a single traffic class, the individual queues got implemented as ring buffers. The structure `queue_t` stores all configuration values to manage a queue. The `read_ptr` defines what entry to read next and `write_ptr` into where to write next. The `length` field defines how many frames fit in the queue.

The queue determines whether an entry is used by checking the `length` value. If the `length` is greater than zero, the entry is in use. If it is zero, the entry is free. The `read_ptr` and `write_ptr` are equal if either the queue is empty or the ring-buffer is full. These two cases can be distinguished via the `length` field. If the `length` field is zero, no frames are in the queue. If it is greater than zero, the queue is full.

7.3.2 Schedule Configuration

A schedule configuration requires the following elements: the duration of the cycle, the GCL and the length of the GCL. In general, the GCL is a list of entries of gate states and a time value. We defined the structure from Listing 7.3 to handle this. We limited the duration of one entry to 56 bits such that this structure fits into 64 bits. The selected time range still allows a timeslot longer than two years, which should exceed all possible requirements for any slot size. With the remaining 8 bits, the structure provides the space required to store the gate states.

For a more straightforward configuration of the *Admin* and *Oper* schedules, we defined the structure `tsn_schedule_config_t` depicted in Listing 7.4. An entry of the schedule requires two fields, one for the gate states (`gate_state`) and one for the duration of this

Listing 7.3: Schedule entry

```
typedef struct
{
  uint64_t gate_state      :8;
  uint64_t slot_duration   :56;
} tsn_gcl_entry_t;
```

Listing 7.4: Schedule configuration

```
typedef struct {
  uint32_t gcl_length;
  tsn_gcl_entry_ptr_t gate_control_list;
  uint32_t cycle_time;
  uint32_t guard_band;
  time_stamp_t update_time;
} tsn_schedule_config_t;
```

slot (`slot_duration`). Additionally, the `cyle_time` member configures the duration the schedule takes. It is common practice in RT networks that a schedule has a short silence period at the end, commonly called a guard band. Therefore, the structure has the member `guard_band` to store the minimum value for this duration. A schedule is valid if the sum of the timeslots and the guard band does not exceed the `base_time` value. The `base_time` member defines when the following schedule shall start. If an update for the schedule is pending, `cycle_extension` defines how long the driver can extend the cycle.

7.3.3 Runtime Configuration

We decided to store these values inside the memory of the Ethernet MAC we are using. Therefore, we designed the `tsn_runtime_config_t` structure. Listing 7.5 shows a reduced version of this structure. All configuration variables regarding a schedule need to exist two times, once in their *Admin* variants and once in their *Oper* variants. The runtime configuration structure references the operational and administrative data as pointers. We decided to use pointers to simplify updating the configurations. Therefore, only the pointer addresses of `admin_schedule` and `oper_schedule` get switched to update the schedule.

The `traffic_queues` member stores the configurations for the individual queues, one queue for each traffic class. Additionally, the member `receive_queue` stores the references to received frames. The `list_pointer` stores which schedule entry is currently active. The `current_gate_state` member stores the transmission gate state of the currently active schedule entry. The `time_interval` member is used to track the remaining time in ticks of this schedule entry.

7.3.4 Executing the Schedule

As already introduced, a schedule consists of multiple time slots, each with a defined time interval. A dedicated timer creates an interrupt when a slot has ended. Therefore, we configured our Patmos target with an additional timer, which only the TSN driver uses. When a time slot has ended, the driver loads the following schedule entry and calculates the timestamp for the next interrupt.

Listing 7.5: Reduced runtime configuration

```
typedef struct {
...
    tsn_schedule_config_ptr_t admin_schedule;
    tsn_schedule_config_ptr_t oper_schedule;
    uint64_t time_interval;
    time_stamp_t cycle_start;
    uint64_t tick_length_ns;
    uint64_t slot_end_time_stamp;
    uint32_t list_pointer;
    uint8_t current_gate_state;
...
} tsn_runtime_config_t;
```

7.3.5 Starting a New Schedule

When the last entry of a TSN schedule finishes its execution, the driver needs to calculate when the following schedule shall start. Therefore, the driver needs to find a point in time which is larger than the current time, and the difference is dividable by the duration of a schedule. In other words, the following equation has to be solved:

$$next\_cycle\_start = cycle\_start + N * schedule\_duration \wedge$$

$$current\_time > cycle\_start + (N-1) * schedule\_duration \wedge$$

$$next\_cycle\_start > current\_time$$

This calculation is a relatively simple operation, as one can calculate N by executing a division, a multiplication, a subtraction and two additions. Unfortunately, this requires 64-bit divisions, which cannot be WCET analysed, as the Patmos compiler uses library functions that are not annotated. Listing 7.6 shows the algorithm behind said calculation avoiding operations that are not WCET analysable.

The algorithm calculates the difference between the current time and the last cycle start. From now on, we reference this as the delta time. Now there are two cases to consider: is delta time larger or equal to one second or less than one second? If the delta time is less than a second, the calculation only uses the nanoseconds portion. Since the nanosecond portion of the timestamp is only 32 bits, we can use division, and the code stays WCET analysable.

If the delta time is larger or equal to one second, the execution of the following steps is required. First, the algorithm calculates how many cycles are required to exceed one second. The amount exceeding one second gets stored, and we reference it from now on as the overhang. Next, the algorithm repeatedly halves the seconds and subtracts the overhang multiplied by the number of seconds we subtract. The algorithm repeats this step until the number of seconds multiplied by the overhang no longer exceeds the nanosecond portion of the delta time.

If the seconds portion of delta time multiplied by the overhang no longer exceeds the nanoseconds portion, this guarantees that the calculated value of delta time will stay positive. Therefore, we approximate the value by repeatedly halving the delta time. Since the multiplication of the overhang and the seconds portion no longer exceed the nanosecond portion, we can execute the prior calculation without halving the seconds first. After this step, the delta time is less than one second.

In this algorithm, we have a loop which requires a corresponding upper bound for the number of execution cycles. We assumed the loop would be executed up to 22 times, corresponding to a time difference of 48 days. Such an extensive delta time would indicate a significant time shift in the network. That indicates a failure of the systems, which requires a restart of the system.

7.4 Further Findings and Reflections

During design and implementation, we came across specific topics out of the scope of this thesis. Further, we found some topics where we have to acknowledge derivations from the standard. For instance, we need to adhere to minimum timeslot durations and consider the limitations of 64-bit operations.

7.4.1 Minimum Timeslot Duration

The standard IEEE 802.1Q [67] defines that the timer ticks shall be in 1 ns second intervals. The used platform has a maximum clock frequency of 80 MHz, which corresponds to a clock period of 12.5 ns. Therefore, we cannot comply with the standard due to limitations in our hardware.

Since we execute the schedule handling inside an interrupt, we also have to consider this. Amongst other interrupt sources, the Ethernet MAC might block the interrupt for too long when handling incoming frames or cleaning up after transmitted ones. Given the platform and its limitations, we decided to use an interval of 1 ms instead.

7.4.2 Schedule Synchronisation

The local clock of the network interface synchronises using gPTP, unlike the system's main clock, which cannot be adapted. Therefore, we use the system clock to generate our timer ticks to execute the schedule. As hardware does not support executing interrupts based on the network clock, it cannot trigger subsequent executions of schedule entries or start the execution of a new cycle. Therefore, we sample the network clock periodically until it reaches the new cycle start time, which limits the accuracy of the execution of the schedule further.

Listing 7.6: Algorithm to calculate the next cycle start

```
time_stamp_t cycle_start = config->cycle_start;
time_interval_t delta;
delta.seconds = netw_clk_time.seconds;
delta.nanoseconds = netw_clk_time.nanoseconds;

delta.seconds -= cycle_start.seconds;
delta.nanoseconds -= cycle_start.nanoseconds;
if(delta.nanoseconds <= -SEC_TO_NS || (delta.nanoseconds< 0 && delta.seconds !=0))
{
  delta.seconds--;
  delta.nanoseconds+=SEC_TO_NS;
}

if(delta.seconds>0)
{
  uint32_t cycles_per_second   = SEC_TO_NS / config->oper_schedule->cycle_time;
  if(cycles_per_second *config->oper_schedule->cycle_time < SEC_TO_NS)
  {
    cycles_per_second++;
  }
  uint32_t overhang_per_second_ns=cycles_per_second*config->oper_schedule->cycle_time;
  overhang_per_second_ns-=  SEC_TO_NS ;
  uint32_t overhang_per_second_ms = overhang_per_second_ns / MS_TO_NS;

  uint32_t overhang_delta_ms=0;
  // Iterate that overhang does not exceed nanosecond portion
  // approximate by halving seconds
  // loopbound 22, assuming clocks are not more than 48 days apart
  #pragma loopbound min 0 max 22
  while (delta.seconds*overhang_per_second_ns > delta.nanoseconds)
  {
    overhang_delta_ms = (delta.seconds>>1) * overhang_per_second_ms;

    delta.seconds -= (delta.seconds >>1)+overhang_delta_ms/SEC_TO_MS;
    delta.nanoseconds -= (overhang_delta_ms % SEC_TO_MS)*MS_TO_NS;

    if(delta.nanoseconds <= -SEC_TO_NS || (delta.nanoseconds< 0 && delta.seconds !=0))
    {
      delta.seconds--;
      delta.nanoseconds+=SEC_TO_NS;
    }

  }
  // overhang does no long exceed nanosecond portion, therefore
  overhang_delta_ms = delta.seconds * overhang_per_second_ms;
  delta.seconds -= delta.seconds + overhang_delta_ms/SEC_TO_MS;
  delta.nanoseconds -= (overhang_delta_ms % SEC_TO_MS)*MS_TO_NS;
}
// delta is now less then a second
uint32_t cycles = delta.nanoseconds/config->oper_schedule->cycle_time;
cycles++;
delta.nanoseconds = config->oper_schedule->cycle_time*cycles-delta.nanoseconds;

cycle_start.seconds = netw_clk_time.seconds;
cycle_start.nanoseconds = netw_clk_time.nanoseconds + delta.nanoseconds ;
```

7.4.3 Network Configuration

As NETCONF adds a significant implementation amount to the thesis, we decided to exclude this topic. NETCONF requires an underlying SSH client-server architecture, which relies on TCP, which is out of the scope of the network stack. Furthermore, SSH requires an encryption library that is not available for the used environment.

7.4.4 Hardware Implementation

Various ways exist to implement the queues for the individual traffic classes, the transmission gates, and the transmission selection. In this thesis, we focused on designing and implementing these features in software. Implementing certain elements in hardware may yield performance improvements. For instance, if the hardware takes over applying the schedule to the transmission gates, this would yield the possibility of having shorter timeslots. Shorter timeslots allow for faster rotation between traffic classes, giving more flexibility for creating schedules and allowing more, e.g., safety-related traffic in between.

It would be attractive to this topic to compare our design against this implementation.

7.4.5 Portability

An essential thought that arose during design and implementation is portability. In general, the used driver allows reuse in driver implementations for other network interfaces. Most elements do not require modifications, but the configuration procedure and buffer descriptor handling need to be adjusted. This aspect shows that most hardware is capable of running a software TSN implementation. Therefore, many already deployed devices can be reused with TSN networks reducing the need for new hardware.

7.5 Evaluation

To evaluate the design of the TSN driver, we did a static WCET analysis of the main functions inside the driver. Additionally, to verify the correct operation, we executed a runtime analysis. This checks if the frames sent out by the driver interface adhere to the defined schedule.

7.5.1 Static WCET Analysis

The main functions used by the network stack are the send function (`tsn_mac_send_nb`) to hand over frames and the receive function (`tsn_mac_reveive_nb`) to read out received frames. Further, the driver requires two functions to keep the Ethernet MAC operational. First, `tsn_mac_update_transmission` handles handing over frames ready to send to the Ethernet MAC and removing already transmitted frames. The function `tsn_mac_update_reception` checks if frames were received, adds them to the receive queue and frees memory not required to store the frame. Further, it allocates new memory in order to store new frames. Finally, the function `handle_tsn_schedule`

Function	WCET	
	cycles	at 80 MHz
`tsn_mac_send_nb ()`	252039	3.15 ms
`buffer_management_malloc ()`	19609	245.11 µs
`mmio_wr_block ()`	97673	1.22 ms
`enqueue_frame ()`	1892	23.65 µs
`buffer_management_free ()`	20734	259.18 µs
`tsn_mac_update_transmission ()`	32128	401.6 µs
`buffer_management_free ()`	20734	259.18 µs
`dequeue_frame ()`	1869	23.36 µs
`tsn_mac_reveive_nb ()`	170498	2.13 ms
`mmio_rd_block ()`	144363	1.8 ms
`buffer_management_free ()`	20734	259.18 µs
`tsn_mac_update_reception ()`	259157	3.24 ms
`buffer_management_realloc ()`	16820	210.25 µs
`buffer_management_malloc ()`	19609	245.11 µs
`handle_tsn_schedule ()`	62799	784.99 µs
`tsn_mac_update_transmission ()`	32128	401.6 µs
`buffer_management_free ()`	20734	259.18 µs
`dequeue_frame ()`	1869	23.36 µs

Table 7.2: WCET analysis values for sending and receiving frames inside the TSN driver

updates the current schedule entry and starts a new cycle. Another task of it is switching between operational and administrational schedules when required.

When we look at `tsn_mac_send_nb`, we see one major contributor to the WCET: `mmio_wr_block`. It gets executed twice, making up a large chunk of the execution time. The significant WCET is because this function copies data to the Ethernet MAC in software. The same applies to `tsn_mac_reveive_nb`, where the major time-consuming step is copying a received frame.

When comparing the two update functions (`tsn_mac_update_transmission` and `tsn_mac_update_reception`), one can see that the reception may take significantly longer. The higher WCET is due to a loop checking each BD reserved for reception. The used configuration uses a maximum of 5 such BDs.

7.5.2 Runtime Analysis

To check if the driver operates correctly, i.e., if frames obey their time slots, we create a small test program which keeps the driver interface fed with frames. It creates a socket for each traffic class and sends 10.000 UDP datagrams for each class. We defined a schedule where each traffic class has a time slot of 1 ms, and the schedule has a total

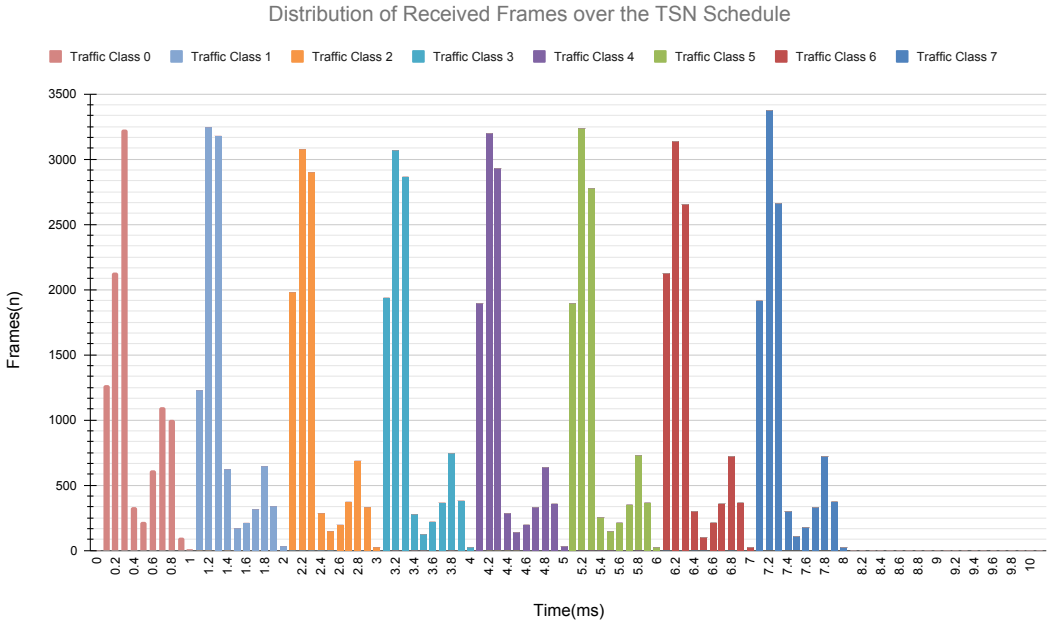


Figure 7.3: Distribution when frames were sent in relation to the schedule

duration of 10 ms. We recorded the reception of the frames using Wireshark. Based on the gathered data, we created a histogram of when our target received the frames in relation to the schedule. Figure 7.3 depicts the results of this measurement.

The histogram from Figure 7.3 depicts how many frames were received during which time interval of the schedule. We can see that all frames related to a single traffic class got received within a one millisecond timespan. This chart shows that the second column is typically higher than the first column, contrary to our initial assumption that the first column should be the highest. This effect is a result of the software implementation of the TSN functionality. One reason is that it takes some time to initiate the transmission of a frame. In addition, the interrupt execution may get delayed, adding some further delay. Multiple messages may get sent in the same time slot depending on how many frames are inside the driver queues. Therefore, there exists a second smaller peak for each traffic class.

7.6 Summary

In this chapter, we introduced the standards TSN builds upon and its main building blocks, see Section 7.1. Based on this, we designed a TSN-aware driver for the used Ethernet MAC. Section 7.3 presents the design, implementation, and Section 7.4 discusses further findings. Section 7.5.1 evaluated how the driver performs in terms of a static WCET analysis. Section 7.5.2 evaluates if the implementation actually adheres to a

TSN schedule. These two sections show that our design and implementation for TSN's scheduled-traffic works. Nonetheless, there is still room for improvement and extension of the feature set, e.g., adding configuration mechanisms based on NETCONF or the credit-based shaper algorithm and improving the size of time slots.

CHAPTER 8

Conclusion

In this chapter, we want to conclude what we achieved in this thesis. In Section 8.1, we revisit our defined research questions and provide our gathered insights for them. Section 8.2 provides an outlook of possible future work regarding the artefacts created by this thesis.

8.1 Gathered Knowledge and Insights

In this thesis, we strived to expand our knowledge about Time-Sensitive Networking (TSN), especially end stations using it. Therefore, we defined in Section 1.1 a set of research questions we wanted to answer. In the following paragraphs, we will revisit the individual questions and summarise the knowledge we gathered during the work on this thesis.

RQ1: How shall a network stack be structured to support real-time and non-real-time traffic? In Chapter 4, we investigated what features a network stack shall provide. Further, we looked into supporting Real-Time (RT) traffic in this chapter. Based on this, we designed a network stack that supports the transmission of mixed-criticality traffic. We employed Worst-Case Execution Time (WCET) analysis for the stack's single elements to guarantee our artefact's deterministic behaviour. We further evaluated our stack using runtime measurements to validate the results of the static code analysis.

In general, the structure of a network stack shall align with a network model like the OSI reference model or the Internet protocol suite. Using such a network model as the underlying structure eases extension of the single layers with further protocols. Further, this separation into the individual layers allows the isolation of processing steps. In addition to the protocols required for operation, a network stack needs to further provide Application Programming Interfaces (APIs) for applications and the system. A

communication interface is required such that the application can communicate with the outside world. A management interface must also be provided to configure network interfaces and other parameters.

The structure of a network stack needs to be designed such that the transmission and reception are WCET analysable. Therefore, mainly static programming patterns have to be used, which comes at the cost of some runtime optimisations like functional pointers. This limitation also prohibits notifying applications about newly received data via callback functions. Therefore, additional buffers are required such that data can be stored until the receiving application processes them. Further, each loop requires bounds so the WCET tooling can analyse the code. If this is not automatically inferable, the loop bounds must be calculated manually and provided to the used WCET tooling.

RQ2: What accuracy can be expected from a clock synchronisation algorithm partly implemented in software? In Chapter 5, we looked into the process of synchronising the clock of our end stations with the network by utilising hardware and software. Based on the requirements of TSN, we selected generalized Precision Time Protocol (gPTP) to syntonise and synchronise the devices. We measured how significantly a clock drifts away from the network without further adjustments. When enabling time synchronisation, we managed to keep the offset from the master port within +/-40 µs.

RQ3: What buffer management algorithm provides the optimal trade-off between time complexity and memory overhead? In Chapter 6, we analysed a set of well-known algorithms for memory management. We investigated the worst case for each algorithm and how it impacts the runtime when requesting or freeing memory. In addition to the execution time analysis, we sketched the worst case for the space requirements of each algorithm.

Based on the gathered knowledge, we concluded that Two-Level Segregated Fit (TLSF) is the best option to realise our RT buffer management for our TSN interface, as it has a constant upper bound for the runtime. For our use case, we further optimised the TLSF to reduce the required memory space to allow the management of a greater number of buffers. We executed a WCET analysis on our implementation to validate the assumed time complexity, which showed that a constant worst-case complexity exists. To validate the correctness of the WCET analysis, we executed runtime tests, which showed that there is no internal element falsifying the WCET.

RQ4: What limitations occur when realising a software-based TSN network interface? In Chapter 7, we designed a driver for a standard Ethernet interface that supports TSN scheduled traffic. The interface supports multiple queues for various types of traffic and uses the buffer management introduced in Chapter 6. The design shows that existing end stations can utilise TSN without needing new interfacing hardware.

However, the implementation revealed several limitations of a software-based approach. We managed to support schedule entries with a length as small as 1 ms. Given the

assumption that the hardware provides all functions, the TSN network interface could run at a significantly higher speed allowing for shorter time slots. Our work shows that large timeslots are the main drawback of a software-based end station. Therefore, the main bottleneck is the used processor, whose clock speed significantly impacts how fast functions can be executed. Since we used a single-core Central Processing Unit (CPU), the available computational power has to be shared with applications running on the system.

A side-effect of the longer minimum timeslot duration is that the cycle intervals also tend to take longer. In other words, a schedule with the same number of entries requires more time using our software-based end station. The standard IEEE 802.1Qbv [8] defines that the duration of a timeslot shall not depend on the network time but only on the local clock. Therefore, only the start of a new cycle is aligned with the network. Since the schedule duration requires additional time, the drift of the local clock may require consideration.

As the efficiency of processing frames is directly dependent on the host's CPU, bandwidth limitations also have to be considered. In hardware, many operations can be executed more efficiently, e.g., updating timeslots and calculating the next cycle start. In software, these steps are performed by the CPU, requiring more time. Therefore, the driver needs to add more safety margin when estimating the transmission time of a frame in order to obey the schedule. These reduced transmission windows inside a timeslot affect the actual usable bandwidth.

8.2 Future Work

In this thesis, we proposed our implementation of a software-based TSN end station. This thesis provides a foundation for further investigations into this topic. The proposed network stack implementation lacks some functions like Transmission Control Protocol (TCP), Internet Protocol (IP) fragmentation and Internet Protocol Version 6 (IPv6) support. These topics are relevant for the general operation and the future of network technologies.

A possible further use for the proposed network stack would be the integration within an Open Platform Communication Unified Architecture (OPC UA) system. For complete operation, OPC UA requires the implementation of TCP. Nonetheless, this network stack guarantees safety-related applications exclusive access to the communication fabric.

In addition, the proposed buffer management implementation does not support multicore or multithreaded access. Therefore, future work could improve the design to allow the usage in multi-consumer and multi-producer environments. Further, there is still potential for improvement in this implementation to provide lower WCET values.

The proposed proof-of-concept suffers from an inefficient copy mechanism. We assume significant performance improvements with technologies like Direct Memory Accesss (DMAs) that handle hardware memory operations. Currently, we handle the checksum

calculation in software, but with dedicated hardware blocks taking over these calculations. This offloading could yield significant runtime improvements.

We designed the proof-of-concept with portability in mind. Therefore, it would be attractive to port this network stack to other chips like RISC-V and evaluate it on them.

The used Ethernet Media Access Control (MAC) provided easy access to control mechanisms to enforce TSN. A survey of other Network Interface Cards (NICs) and driver interfaces would be interesting to determine if this approach is easily portable. In addition, it would be interesting to see how to implement a driver for a NIC that already provides queues and traffic gates defined by TSN.

List of Figures

List of Tables

Listings

Glossary

ARM® ARM is a processor architecture mostly used for mobile devices and embedded devices. ARM stands for "Advanced RISC Machines". 7

AUTOSAR A standardisation consortium for automative manufacturers and suppliers. 10

Bluetooth A wireless protocol used for Personal Area Networks, defined in IEEE 802.15.1. 25

Boundary-Tag A small data structure at the beginning or the end of memory area, signalling its boundaries. 46, 60, 61, 66

C is a general purpose programming language, often used embedded systems or programming of Operating Systems. 7

credit-based shaper algorithm The credit-based shaper algorithm, is a traffic selection algorithm, which defines that queued frames shall only be sent if the transmission rate was not exceeded. 76, 78

end station A device attached to a local area network (LAN) or metropolitan area network (MAN), which acts as a source of and/or destination for data traffic carried on the LAN or MAN. [74]. 2, 3, 5, 10, 101

end-system A system attached to a network that is an initial source or a final destination of packets transmitted across that network. [74]. 2, 4, 8, 10

Enhanced Transmission Selection Algorithm The Enhanced Transmission Selection algorithm, is a traffic selection algorithm, which defines that queued frames shall only be sent if the allocated bandwidth was not exceeded. 76, 78

Ethernet Is a protocol which operates on the layer 2 of the OSI reference model. It is defined standard IEEE 802.3 [2]. 1, 7–10, 16, 18–20, 24, 25, 27–31, 35, 41, 42, 61–63, 75, 76, 78, 81, 82, 86, 88, 97

Free-List A list representing free/unallocated buffer entries. 46, 50–62, 64, 66, 69, 70, 91

grandmaster The time-aware system that contains the best clock, as determined by the best master clock algorithm (BMCA), in the generalized precision time protocol (gPTP) domain. [26]. 8, 35, 37–40

Industry 4.0 The fourth industrial revoltuion with focus on interconnectivity and smart automation. 2, 9, 10

Internet protocol suite a model for the protocol layers of the internet defined by the Internet Engineering Task Force (IETF). 15–18, 20, 91

IT/OT convergence The IT/OT convergence defines the process of integrating Information Technology (IT) into systems with Operational Technology (OT). 1, 8, 9

Java java is java. 46

Linux Linux is the superset of open-source operating systems based on the Linux kernel. 17, 24, 26

Linux PTP Linux PTP is an open-source project which implements the Precision Time Protocol (PTP) for Linux-based Operating Systems (OSs), see `https://linuxptp.sourceforge.net/` last accessed 9.12.2022. 42

llvm Is a widely used compiler and toolchain ecosystem. 7

middleware A software block that abstracts the communication from applicationss. 9

OSI reference model Open Systems Interconnection model, a model which standardises functions between telecommunication and operating systems. 15–18, 91, 99

Patmos Patmos is a time-predictable very-long instruction-word (VLIW) processor. It is intended as a processor for embedded real-time systems. [20]. xv, 2, 3, 5, 7, 10, 28, 68, 83, 91

Profibus Process Field Bus, is a field bus developed, by the German department of education and research, for industrial automation. 1

ProfiNET ProfiNET is a Ethernet based Standard from Siemens, which is mainly used in industrial auotmation. 7, 10

RISC-V RISC-V is an Instruction Set Architecture (ISA) thas was designed within a project of the UC Berekly.. 7

router A router is a device which is located on OSI-Layer 3. Routers handle the routing of packets between networks. 2

RX Short form for reception. 10, 41, 42, 86

SSH Secure Shell provides a protection layer to access services securly on unsecure networks. Typically used for remote command execution or login into devices. In general, SSH can be added to any network service.. 16, 76

strict priority algorithm The strict priority algorithm, is a traffic selection algorithm, which defines that the frames queued in the highest priority traffic class shall be sent first. 78

switch A switch has multiple network ports, where each port provides bridge functionality. 2, 9, 18, 76

time-aware bridge A Bridge that is capable of communicating synchronized time received on one port to other ports, using the IEEE 802.1AS protocol. [26]. 35

time-aware end station An end station that is capable of acting as the source of synchronized time on the network, or destination of synchronized time using the IEEE 802.1AS protocol, or both. [26]. 35

time-aware system A time-aware Bridge or a time-aware end station. [26]. 35, 37

traffic class A classification used to expedite transmission of frames generated by critical or time-sensitive services. Traffic classes are numbered from zero through N-1, where N is the number of outbound queues associated with a given Bridge port, and 1 <= N <= 8, and each traffic class has a one-to-one correspondence with a specific outbound queue for that port. Traffic class 0 corresponds to nonexpedited traffic; nonzero traffic classes correspond to expedited classes of traffic. A fixed mapping determines, for a given priority associated with a frame and a given number of traffic classes, what traffic class will be assigned to the frame. [67]. 8, 10, 76, 78, 81, 82, 87, 88, 101

transmission gate A gate that connects or disconnects the transmission selection function of the forwarding process from the queue, allowing or preventing it from selecting frames from that queue. The gate has two states, Open and Closed. [67]. 10, 78–80, 86

TTCAN Time Triggered Controller Area Network is a time aware extension to the well known Controller Area Network (CAN) bus.. 10

TX Short form for transmission. 10, 41, 42

VLAN The closure of a set of MAC Service Access Points (MSAPs) such that a data request in one MSAP in the set is expected to result in a data indication in another MSAP in the set. [67]. 19, 29, 31, 76

Wireshark is a packet analyzer, which is often used for troubleshooting and network analysis. 87

x86 Is a Complex Instruction Set Computer (CISC) based CPU architecture from Intel.. 7

XML Extensible Markup Language, a markup-language which defines rules for encoding to be human and machine-readable. 76

YANG YANG (Yet another next Generation) is a modelling language, in combination with Network Configuration (NETCONF) it is used for modelling configurations.. 3, 76

Acronyms

API Application Programming Interface. 23, 24

ARP Address Resolution Protocol. 16, 18, 19, 29, 30, 63

AVB Audio Video Bridging. 75

BCET Best-Case Execution Time. 6

BD Buffer Descriptor. 20, 86

BE Best Effort. 8, 76

BMCA Best Master Clock Algorithm. 35, 37–39, 41, 91

CAN Controller Area Network. 1, 101

CAS Compare-and-Swap. 3, 68

CISC Complex Instruction Set Computer. 7, 101

COTS Commercial of the Shelf. 7

CPS Cyber-Physical System. 5

CPU Central Processing Unit. 7, 20, 28, 64, 68, 101

CQF Cyclic queuing and forwarding. 3

CRC Cyclic Redundancy Check. 19, 24, 30

CSMA/CD Carrier-Sense Multiple Access with Collision Detection. 8, 18, 76

DDS Data Distribution Service. 9

defaultDS Default Data Set. 37, 38

DHCP Dynamic Host Configuration Protocol. 76

DMA Direct Memory Access. 28

DNS Domain Name System. 16, 76

DSA Dynamic Storage Allocation. 4, 45–50, 58, 60, 73, 93

E2E End-to-End. 16, 21, 35

ERP Enterprise Resource Planning. 2

EUI Extended Unique Identifier. 19, 40

FCS Frame Check Sequence. 19, 30, 31

FLI First Level Index. 56, 57, 62, 64–66, 70–72, 97

FPGA Field-Programmable Gate Array. 8–10

FPU Floating Point Unit. 64

FRER Frame Replication and Elimination for Reliability. 3

Gbit/s Gigabit per second. 1, 18

GCL Gate Control List. 79–81

GNSS Global Navigation Satellite System. 34, 38

GPS Global Positioning System. 34

gPTP generalized Precision Time Protocol. xv, 4, 8, 11, 33, 35–37, 39, 41, 75, 84, 91

I2C Inter-Integrated Circuit. 20

ICMP Internet Control Message Protocol. 16, 21, 27, 31

IEEE Institute of Electrical and Electronics Engineers. 1, 8, 18, 35, 75, 77, 81, 84, 93, 99

IETF Internet Engineering Task Force. 16, 22, 100

IIoT Industrial Internet of Things. 1, 2, 9

IO Input/Output. 20, 68

IoT Internet of Things. 1, 9

IP Internet Protocol. 15, 16, 20–22, 25, 27, 29, 31, 76

IP-Core Intellectual Property Core. 8, 9

IPv4 Internet Protocol Version 4. 19–21, 27, 29–31

IPv6 Internet Protocol Version 6. 20, 21, 27

ISA Instruction Set Architecture. 7, 100

IT Information Technology. 1, 2, 7–9, 100

ITU-T International Telecommunication Union Telecommunication Standardization Sector. 15

LAN Local Area Network. 19, 34

LL/SC Load-Link/Store-Conditional. 3, 68

MAC Media Access Control. 8, 19, 20, 27–30, 40–42, 61–63, 81, 82, 86, 88, 97, 101

Mbit/s Megabit per second. 1, 18, 42

MCU Microcontroller Unit. 35

MII Media Independent Interface. 10, 41, 42

MQTT Message Queue Telemetry Transport. 9

MTU Maximum Transmission Unit. 21, 27

NETCONF Network Configuration. 3, 8, 76, 86, 88, 102

NIC Network Interface Card. 9, 18, 24, 27, 35

NTP Network Time Protocol. 34

OPC UA Open Platform Communication Unified Architecture. 9

OS Operating System. 17, 45, 58

OT Operational Technology. 1, 2, 8, 9, 100

P2P Peer-to-Peer. 35, 36

PCIe Peripheral Component Interconnect Express. 20

PCR Path Control and Reservation. 3

PHY Physical-Layer. 10, 28, 41

PLC Programmable Logic Controller. 2

PSFP Per-Stream Filtering and Policing. 3

PTP Precision Time Protocol. 3, 8, 10, 34, 35, 37, 39, 41, 42

RAM Random Access Memory. 45

RFC Request for Comments. 16, 19, 21, 22

RISC Reduced Instruction Set Computer. 7

ROS Robot Operating System. 9

RPC Remote Procedure Call. 76

RT Real-Time. 4, 9, 10, 15, 25–29, 31, 48, 68, 75, 76, 81, 91

RTC Real-Time Clock. 41

RTOS Real-Time Operating System. 41

RTS Real-Time System. xv, 3–6, 10, 15, 22, 34, 45, 68

SCADA Supervisory Control and Data Acquisition. 2

SDN Software-Defined Networking. 10

SLI Second Level Index. 57, 62, 64–66, 70–72, 97

SNMP Simple Network Management Protocol. 76

SoC System on a Chip. 20, 35

SPI Serial Peripheral Interface. 20

SRP Stream Reservation Protocol. 3, 8

SSH Secure Shell. 16, 76, 86, *Glossary:* SSH

TCP Transmission Control Protocol. 16, 22, 23, 25, 27, 31, 86

TDMA Time-Division Multiple Access. 8

TLSF Two-Level Segregated Fit. 55–64, 67, 68, 73, 91, 93, 97

TSA Transmission Selection Algorithm. 78

TSN Time-Sensitive Networking. xv, xvi, 1–5, 7–11, 20, 25, 33, 35, 41, 75–81, 83, 86–88, 91, 93

TTCAN Time Triggered Controller Area Network. 10, *Glossary:* TTCAN

TTEthernet Time-Triggered Ethernet. 7, 8, 10

TTP Time-Triggered Protocol. 10

UDP User Datagram Protocol. 16, 22, 25, 27, 29–31, 34, 87

UTC Universal Time Coordinated. 34

VHDL Very High Speed Integrated Circuit Hardware Description Language. *Glossary:* VHDL

VL Virtual Link. 8

WCET Worst-Case Execution Time. xv, 2–4, 6, 7, 9, 10, 15, 27–31, 50, 69, 72, 73, 83, 86–88, 91, 93

WLAN Wireless Local Area Network. 18, 25

Bibliography

[1] "PROFIBUS is a standardized, open, digital communications system for all areas of application in manufacturing and process automation.." https://www.profibus.com/technology/profibus. Accessed: 2021-11-18.

[2] "IEEE Standard for Ethernet," *IEEE Std 802.3-2018 (Revision of IEEE Std 802.3-2015)*, pp. 1–5600, 2018.

[3] "What is IT/OT convergence? Everything you need to know." https://searchitoperations.techtarget.com/definition/IT-OT-convergence. Accessed: 2021-11-18.

[4] F. Bonomi, R. Milito, J. Zhu, and S. Addepalli, "Fog Computing and Its Role in the Internet of Things," in *Proceedings of the First Edition of the MCC Workshop on Mobile Cloud Computing*, MCC '12, (New York, NY, USA), pp. 13–16, Association for Computing Machinery, 2012.

[5] "Hirrschmann White Paper: TSN – Time Sensitive Networking." https://www.belden.com/dfsmedia/f1e38517e0cd4caa8b1acb6619890f5e/7897-source. Accessed: 2021-11-18.

[6] "PROFINET is the innovative open standard for Industrial Ethernet. It satisfies all requirements of automation technology.." https://www.profibus.com/technology/profinet. Accessed: 2021-11-18.

[7] "Ethernet Powerlink is a network technology based on Ethernet with focus real-time operation and safety for industrial automation.." https://www.ethernet-powerlink.org/powerlink/technology. Accessed: 2022-12-20.

[8] "IEEE Standard for Local and metropolitan area networks – Bridges and Bridged Networks - Amendment 25: Enhancements for Scheduled Traffic," *IEEE Std 802.1Qbv-2015 (Amendment to IEEE Std 802.1Q— as amended by IEEE Std 802.1Qca-2015, IEEE Std 802.1Qcd-2015, and IEEE Std 802.1Q—/Cor 1-2015)*, pp. 1–57, March 2016.

[9] W. Quan, W. Fu, J. Yan, and Z. Sun, "OpenTSN: an open-source project for time-sensitive networking system development," *CCF Transactions on Networking*, vol. 3, 08 2020.

[10] A. C. T. d. Santos, B. Schneider, and V. Nigam, "TSNSCHED: Automated Schedule Generation for Time Sensitive Networking," in *2019 Formal Methods in Computer Aided Design (FMCAD)*, pp. 69–77, 2019.

[11] N. Reusch, L. Zhao, S. S. Craciunas, and P. Pop, "Window-Based Schedule Synthesis for Industrial IEEE 802.1Qbv TSN Networks," in *2020 16th IEEE International Conference on Factory Communication Systems (WFCS)*, pp. 1–4, 2020.

[12] B. Wang, F. Luo, Y. Jin, Z. Fang, and Q. Li, "Design of TSN-based Ethernet Driver Working Model for Time-aware Scheduling," in *2021 International Conference on Intelligent Technology and Embedded Systems (ICITES)*, pp. 157–163, 2021.

[13] "IEEE Standard for Local and metropolitan area networks–Frame Replication and Elimination for Reliability," *IEEE Std 802.1CB-2017*, pp. 1–102, Oct 2017.

[14] M. Pahlevan and R. Obermaisser, "Redundancy Management for Safety-Critical Applications with Time Sensitive Networking," in *2018 28th International Telecommunication Networks and Applications Conference (ITNAC)*, pp. 1–7, 2018.

[15] "IEEE Standard for Local and metropolitan area networks— Bridges and Bridged Networks - Amendment 24: Path Control and Reservation," *IEEE Std 802.1Qca-2015 (Amendment to IEEE Std 802.1Q-2014 as amended by IEEE Std 802.1Qcd-2015 and IEEE Std 802.1Q-2014/Cor 1-2015)*, pp. 1–120, March 2016.

[16] "IEEE Standard for Local and metropolitan area networks–Bridges and Bridged Networks–Amendment 29: Cyclic Queuing and Forwarding," *IEEE 802.1Qch-2017 (Amendment to IEEE Std 802.1Q-2014 as amended by IEEE Std 802.1Qca-2015, IEEE Std 802.1Qcd(TM)-2015, IEEE Std 802.1Q-2014/Cor 1-2015, IEEE Std 802.1Qbv-2015, IEEE Std 802.1Qbu-2016, IEEE Std 802.1Qbz-2016, and IEEE Std 802.1Qci-2017)*, pp. 1–30, June 2017.

[17] "IEEE Standard for Local and metropolitan area networks–Bridges and Bridged Networks–Amendment 28: Per-Stream Filtering and Policing," *IEEE Std 802.1Qci-2017 (Amendment to IEEE Std 802.1Q-2014 as amended by IEEE Std 802.1Qca-2015, IEEE Std 802.1Qcd-2015, IEEE Std 802.1Q-2014/Cor 1-2015, IEEE Std 802.1Qbv-2015, IEEE Std 802.1Qbu-2016, and IEEE Std 802.1Qbz-2016)*, pp. 1–65, Sep. 2017.

[18] "Source code repository for this thesis.." `https://git.auto.tuwien.ac.at/rt-ua/libnetwork`. Accessed: 2022-12-22.

[19] "T-CREST: Time-predictable-multicore Architecture for embedded Systems." `http://www.t-crest.org/`. Accessed: 2021-11-18.

[20] M. Schoeberl, S. Abbaspour, B. Akesson, N. Audsley, R. Capasso, J. Garside, K. Goossens, S. Goossens, S. Hansen, R. Heckmann, S. Hepp, B. Huber, A. Jordan, E. Kasapaki, J. Knoop, Y. Li, D. Prokesch, W. Puffitsch, P. Puschner, A. Rocha,

C. Silva, J. Sparsø, and A. Tocchi, "T-CREST: Time-predictable multi-core architecture for embedded systems," *Journal of Systems Architecture*, vol. 61, no. 9, pp. 449–471, 2015.

[21] H. Kopetz, *Real-Time Systems - Design Principles for Distributed Embedded Applications*. Real-Time Systems Series, Springer, 2011.

[22] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, F. Mueller, I. Puaut, P. Puschner, J. Staschulat, and P. Stenström, "The Worst-Case Execution-Time Problem—Overview of Methods and Survey of Tools," *ACM Trans. Embed. Comput. Syst.*, vol. 7, may 2008.

[23] S. Hepp, B. Huber, J. Knoop, D. Prokesch, and P. P. Puschner, "The platin tool kit-the T-CREST approach for compiler and WCET integration," in *Proceedings 18th Kolloquium Programmiersprachen und Grundlagen der Programmierung, KPS*, pp. 5–7, 2015.

[24] *SAE AS6802 - Time-Triggered Ethernet*, nov 2016.

[25] "IEEE Standard for a Precision Clock Synchronization Protocol for Networked Measurement and Control Systems," *IEEE Std 1588-2019 (Revision ofIEEE Std 1588-2008)*, pp. 1–499, 2020.

[26] "IEEE Standard for Local and Metropolitan Area Networks–Timing and Synchronization for Time-Sensitive Applications," *IEEE Std 802.1AS-2020 (Revision of IEEE Std 802.1AS-2011)*, pp. 1–421, 2020.

[27] S. S. Craciunas, R. Serna Oliver, and W. Steiner, "Demo Abstract: Slate XNS–An Online Management Tool for Deterministic TSN Networks," in *2018 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pp. 103–104, 2018.

[28] S. Gent, P. Gutiérrez Peón, T. Frühwirth, and D. Etz, "Hosting functional safety applications in factory networks through Time-Sensitive Networking," in *2020 25th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*, vol. 1, pp. 230–237, 2020.

[29] D. Etz, T. Frühwirth, and W. Kastner, "Flexible Safety Systems for Smart Manufacturing," in *2020 25th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*, vol. 1, pp. 1123–1126, 2020.

[30] P. Denzler, T. Frühwirth, A. Kirchberger, M. Schoeberl, and W. Kastner, "Experiences from Adjusting Industrial Software for Worst-Case Execution Time Analysis," in *2021 IEEE 24th International Symposium on Real-Time Distributed Computing (ISORC)*, pp. 62–70, 2021.

[31] P. Denzler, T. Frühwirth, A. Kirchberger, M. Schoeberl, and W. Kastner, "Static Timing Analysis of OPC UA PubSub," in *2021 17th IEEE International Conference on Factory Communication Systems (WFCS)*, pp. 167–174, 2021.

[32] S. Profanter, A. Tekat, K. Dorofeev, M. Rickert, and A. Knoll, "OPC UA versus ROS, DDS, and MQTT: Performance Evaluation of Industry 4.0 Protocols," in *2019 IEEE International Conference on Industrial Technology (ICIT)*, pp. 955–962, 2019.

[33] L. Silva, P. Pedreiras, P. Fonseca, and L. Almeida, "On the adequacy of SDN and TSN for Industry 4.0," in *2019 IEEE 22nd International Symposium on Real-Time Distributed Computing (ISORC)*, pp. 43–51, 2019.

[34] S. B. H. Said, Q. H. Truong, and M. Boc, "SDN-Based Configuration Solution for IEEE 802.1 Time Sensitive Networking (TSN)," *SIGBED Rev.*, vol. 16, p. 27–32, feb 2019.

[35] E. Kyriakakis, J. Sparsø, and M. Schoeberl, "Hardware Assisted Clock Synchronization with the IEEE 1588-2008 Precision Time Protocol," in *Proceedings of the 26th International Conference on Real-Time Networks and Systems*, RTNS '18, (New York, NY, USA), p. 51–60, Association for Computing Machinery, 2018.

[36] T. Frühwirth, W. Steiner, and B. Stangl, "TTEthernet SW-based end system for AUTOSAR," in *10th IEEE International Symposium on Industrial Embedded Systems (SIES)*, pp. 1–8, 2015.

[37] E. Kyriakakis, M. Lund, L. Pezzarossa, J. Sparsø, and M. Schoeberl, "A time-predictable open-source TTEthernet end-system," *Journal of Systems Architecture*, vol. 108, p. 101744, 2020.

[38] B. J. Oates, *Researching information systems and computing*. Sage, 2005.

[39] S. T. March and G. F. Smith, "Design and natural science research on information technology," *Decision Support Systems*, vol. 15, no. 4, pp. 251–266, 1995.

[40] P. Checkland, "Soft systems methodology: a thirty year retrospective," *Systems research and behavioral science*, vol. 17, no. S1, pp. S11–S58, 2000.

[41] A. R. Hevner, S. T. March, J. Park, and S. Ram, "Design science in information systems research," *MIS Quarterly*, vol. 28, no. 1, pp. 75–105, 2004.

[42] J. Hughes and T. Wood-Harper, "Systems development as a research act," *Journal of Information Technology*, vol. 14, no. 1, pp. 83–94, 1999.

[43] I. Standardization, "ISO/IEC 7498-1: 1994 information technology–open systems interconnection–basic reference model: The basic model," *International Standard ISOIEC*, vol. 74981, p. 59, 1996.

[44] R. T. Braden, "Requirements for Internet Hosts - Communication Layers." RFC 1122, Oct. 1989.

[45] R. T. Braden, "Requirements for Internet Hosts - Application and Support." RFC 1123, Oct. 1989.

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

[47] "An Ethernet Address Resolution Protocol: Or Converting Network Protocol Addresses to 48.bit Ethernet Address for Transmission on Ethernet Hardware." RFC 826, Nov. 1982.

[48] J. Corbet, A. Rubini, and G. Kroah-Hartman, *Linux Device Drivers, Third Edition*. O'Reilly Media, Inc., 3 ed., 2005.

[49] "Internet Protocol." RFC 791, Sept. 1981.

[50] B. Hinden and D. S. E. Deering, "Internet Protocol, Version 6 (IPv6) Specification." RFC 2460, Dec. 1998.

[51] D. S. E. Deering and B. Hinden, "Internet Protocol, Version 6 (IPv6) Specification." RFC 8200, July 2017.

[52] "Internet Control Message Protocol." RFC 792, Sept. 1981.

[53] "User Datagram Protocol." RFC 768, Aug. 1980.

[54] "Transmission Control Protocol." RFC 793, Sept. 1981.

[55] M. Kerrisk, *The Linux Programming Interface: A Linux and UNIX System Programming Handbook*. USA: No Starch Press, 1st ed., 2010.

[56] A. Silberschatz, P. B. Galvin, and G. Gagne, *Operating System Concepts Essentials, 2nd Edition*. Wiley, 2014.

[57] M. A. Awais, "Memory management: Challenges and techniques for traditional memory allocation algorithms in relation with today's real time needs," *Advances in Computer Science: an International Journal*, vol. 5, no. 2, pp. 22–27, 2016.

[58] I. Puaut, "Real-time performance of dynamic memory allocation algorithms," in *Proceedings 14th Euromicro Conference on Real-Time Systems. Euromicro RTS 2002*, pp. 41–49, 2002.

[59] M. Tadman, "Fast-fit: A new hierarchical dynamic storage allocation technique," *Master's thesis*, 1978.

[60] R. Bayer and E. McCreight, *Organization and Maintenance of Large Ordered Indexes*, pp. 245–262. Berlin, Heidelberg: Springer Berlin Heidelberg, 2002.

[61] K. C. Knowlton, "A fast storage allocator," *Communications of the ACM*, vol. 8, no. 10, pp. 623–624, 1965.

[62] T. Ogasawara, "An algorithm with constant execution time for dynamic storage allocation," in *Proceedings Second International Workshop on Real-Time Computing Systems and Applications*, pp. 21–25, 1995.

[63] M. Masmano, I. Ripoll, A. Crespo, and J. Real, "TLSF: a new dynamic memory allocator for real-time systems," in *Proceedings. 16th Euromicro Conference on Real-Time Systems, 2004. ECRTS 2004.*, pp. 79–88, 2004.

[64] M. Masmano, I. Ripoll, P. Balbastre, and A. Crespo, "A constant-time dynamic storage allocator for real-time systems," *Real-Time Systems*, vol. 40, no. 2, pp. 149–179, 2008.

[65] H. S. Warren, *Hacker's Delight*. Addison-Wesley Professional, 2nd ed., 2012.

[66] "IEEE Standard for Local and Metropolitan Area Networks–Audio Video Bridging (AVB) Systems," *IEEE Std 802.1BA-2021 (Revision of IEEE Std 802.1BA-2011)*, pp. 1–45, 2021.

[67] "IEEE Standard for Local and Metropolitan Area Network–Bridges and Bridged Networks," *IEEE Std 802.1Q-2018 (Revision of IEEE Std 802.1Q-2014)*, pp. 1–1993, July 2018.

[68] R. Enns, M. Björklund, A. Bierman, and J. Schönwälder, "Network Configuration Protocol (NETCONF)." RFC 6241, June 2011.

[69] M. Björklund, "The YANG 1.1 Data Modeling Language." RFC 7950, Aug. 2016.

[70] "IEEE Standard for Local and metropolitan area networks - Station and Media Access Control Connectivity Discovery," *IEEE Std 802.1AB-2016 (Revision of IEEE Std 802.1AB-2009)*, pp. 1–146, March 2016.

[71] "IEEE Standard for Local and metropolitan networks–Station and Media Access Control Connectivity Discovery Amendment 1: YANG Data Model," *IEEE Std 802.1ABcu-2021 (Amendment to IEEE Std 802.1AB-2016)*, pp. 1–55, 2022.

[72] "IEEE Standard for Local and metropolitan area networks– Station and Media Access Control Connectivity Discovery Amendment 2: Support for Multiframe Protocol Data Units," *IEEE Std 802.1ABdh-2021 (Amendment to IEEE Std 802.1AB-2016 as amended by IEEE Std 802.1ABcu-2021)*, pp. 1–56, 2022.

[73] "IEEE Standard for Local and metropolitan area networks – Link Aggregation," *IEEE Std 802.1AX-2014 (Revision of IEEE Std 802.1AX-2008)*, pp. 1–344, Dec 2014.

[74] "IEEE Standard for Local and metropolitan area networks–Frame Replication and Elimination for Reliability," *IEEE Std 802.1CB-2017*, pp. 1–102, Oct 2017.

[75] "IEEE Standard for Local and metropolitan area networks – Frame Replication and Elimination for Reliability - Amendment 1: Information Model, YANG Data Model, and Management Information Base Module," *IEEE Std 802.1CBcv-2021 (Amendment to IEEE Std 802.1CB-2017)*, pp. 1–157, 2022.

[76] "IEEE Standard for Local and metropolitan area networks – Time-Sensitive Networking for Fronthaul," *IEEE Std 802.1CM-2018*, pp. 1–62, June 2018.

[77] "IEEE Standard for Local and metropolitan area networks – Time-Sensitive Networking for Fronthaul - Amendment 1: Enhancements to Fronthaul Profiles to Support New Fronthaul Interface, Synchronization, and Syntonization Standards," *IEEE Std 802.1CMde-2020 (Amendment to IEEE Std 802.1CM-2018*, pp. 1–35, 2020.

[78] "IEEE Standard for Local and Metropolitan Area Networks–Link-local Registration Protocol," *IEEE Std 802.1CS-2020*, pp. 1–151, 2021.

[79] "IEEE Standard for Local and metropolitan area networks – Bridges and Bridged Networks – Amendment 26: Frame Preemption," *IEEE Std 802.1Qbu-2016 (Amendment to IEEE Std 802.1Q-2014)*, pp. 1–52, Aug 2016.

[80] "IEEE Standard for Local and Metropolitan Area Networks–Bridges and Bridged Networks – Amendment 31: Stream Reservation Protocol (SRP) Enhancements and Performance Improvements," *IEEE Std 802.1Qcc-2018 (Amendment to IEEE Std 802.1Q-2018 as amended by IEEE Std 802.1Qcp-2018)*, pp. 1–208, Oct 2018.

[81] "IEEE Standard for Local and metropolitan area networks–Bridges and Bridged Networks–Amendment 30: YANG Data Model," *IEEE Std 802.1Qcp-2018 (Amendment to IEEE Std 802.1Q-2018)*, pp. 1–93, 2018.

[82] "IEEE Standard for Local and Metropolitan Area Networks–Bridges and Bridged Networks - Amendment 34:Asynchronous Traffic Shaping," *IEEE Std 802.1Qcr-2020 (Amendment to IEEE Std 802.1Q-2018 as amended by IEEE Std 802.1Qcp-2018, IEEE Std 802.1Qcc-2018, IEEE Std 802.1Qcy-2019, and IEEE Std 802.1Qcx-2020)*, pp. 1–151, 2020.

[83] "IEEE Standard for Local and Metropolitan Area Networks–Bridges and Bridged Networks Amendment 33: YANG Data Model for Connectivity Fault Management," *IEEE Std 802.1Qcx-2020 (Amendment to IEEE Std 802.1Q-2018 as amended by IEEE Std 802.1Qcp-2018, IEEE Std 802.1Qcc-2018, and IEEE Std 802.1Qcy-2019)*, pp. 1–123, 2020.