Institute of Parallel and Distributed Systems

University of Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Bachelorarbeit

Investigation and Verification of Real-time Schedules via Simulation

Lars Philipp

Course of Study:	Softwaretechnik
Examiner:	Prof. Dr. Christian Becker
Supervisor:	Heiko Geppert, M.Sc.
Commenced:	January 24, 2024
Completed:	July 24, 2024

Abstract

With the rise in popularity of Cyber-Physical Systems (CPSs) in various domains such as the Internet of Things (IoT) and autonomous vehicles, the demand for real-time systems especially with deterministic real-time communication, has significantly grown. Bounded latencies are a crucial part of the safety of CPSs, maintained through deterministic real-time communications. The Institute of Electrical and Electronics Engineers (IEEE) recognized the rise in popularity of such real-time systems with deterministic communication, resulting in a set of standards known as Time-Sensitive Networking (TSN). Although this standard establishes the technological infrastructure for deterministic communication on Ethernet networks, it does not specify the computation of schedules for TSN networks.

This led to the development of scheduling algorithms for TSN networks becoming a major field of research. A critical step prior to the deployment of these algorithms is to investigate and verify them. In this study, we investigate and verify the scheduling algorithms through a simulation of the computed schedule on TSN networks. A simulation offers an easy and cost-effective way to verify the scheduling algorithms.

We utilize the discrete event simulator OMNeT++ (Objective Modular Network Test-bed in C++) with the INET framework for the simulation of the scheduling algorithms. In particular, we simulate schedules generated with the Greedy Flow Heap (GFH) algorithm. This algorithm uses a conflict-graph-approach to compute the schedules. In order to facilitate the simulation process, we have developed a pipeline that effectively transforms the computed schedules of the algorithm into a fully operational simulation. In our evaluation, we successfully compared the computed schedules of the scheduling algorithm with the simulation results. Furthermore, we show the limitations of the scheduling algorithm, which are primarily attributable to the existing implementation of the algorithm.

Kurzfassung

Mit der zunehmenden Beliebtheit von Cyber-Physical Systems (CPSs) in verschiedenen Bereichen wie dem Internet der Dinge (IoT) und autonomen Fahrzeugen, ist die Nachfrage nach Echtzeitsystemen, insbesondere mit deterministischer Echtzeitkommunikation stark gewachsen. Begrenzte Netzwerklatenzen sind ein entscheidender Teil der Sicherheit von CPSs, die durch die deterministischer Echtzeitkommunikation eingehalten werden. Das Institute of Electrical and Electronics Engineers (IEEE) erkannte die zunehmende Beliebtheit solcher Echtzeitsysteme mit deterministischer Kommunikation und führte eine Reihe von Standards ein, die als Time-Sensitive Networking (TSN) bekannt sind. Obwohl diese Standards die technologische Infrastruktur für die deterministische Kommunikation in Ethernet-Netzwerken festlegt, spezifizieren sie nicht die Berechnung von Schedules für TSN-Netzwerke.

Dies führte dazu, dass die Entwicklung von Scheduling-Algorithmen für TSN-Netzwerke zu einem wichtigen Forschungsgebiet wurde. Ein entscheidender Schritt vor dem Einsatz dieser Algorithmen in echten Netzwerken besteht darin, sie zu überprüfen. In dieser Studie untersuchen und verifizieren wir die Scheduling-Algorithmen mittels einer Simulation des von ihnen berechneten Zeitplans in TSN-Netzwerken. Die Simulation bietet eine einfache und kostengünstige Möglichkeit, die Scheduling-Algorithmen zu überprüfen.

Für die Simulation der Scheduling-Algorithmen nutzen wir den Netzwerksimulator OMNeT++ (Objective Modular Network Test-bed in C++) mit dem INET-Framework. Insbesondere simulieren wir Schedules, die mit dem Greedy Flow Heap (GFH)-Algorithmus erstellt wurden. Dieser Algorithmus verwendet einen Konfliktgraphen-Ansatz zur Berechnung der Schedules. Um den Simulationsprozess zu erleichtern, haben wir eine Pipeline entwickelt, die die berechneten Schedules des Algorithmus effektiv in eine voll funktionsfähige Simulation umwandelt. In unserer Auswertung haben wir die berechneten Zeitpläne des Scheduling-Algorithmus erfolgreich überprüft. Darüber hinaus zeigen wir die Einschränkungen des Scheduling-Algorithmus auf, die in erster Linie auf die bestehende Implementierung des Algorithmus zurückzuführen sind.

Contents

List of Figures

List of Tables

List of Listings

List of Algorithms

Acronyms

CBS	credit-based shaper.
CP	Constraint Programming.
CPSs	Cyber-Physical Systems.
DA/IRS	DoC-aware iterative routing and scheduling.
DAMR	DoC-Aware Multipath Routing.
DASP	DoC-aware streams partitioning.
DoC	Degree of Conflict.
GCL	Gate Control List.
GFH	Greedy Flow Heap Heuristic.
HERMES	heuristic multi-queue scheduler.
ICS	independent colorful set.
IDE	Integrated Development Environment.
IEEE	Institute of Electrical and Electronics Engineers.
IIS	iterated integer linear programming-based scheduling.
ILP	Integer Linear Programming.
IoT	Internet of Things.
IPv4	Internet Protocol version 4.
IS	independent set.
IT	Information Technology.
MAC	Media Access Control.
MTU	maximum transmission unit.
NED	network definition file.
NP	nondeterministic polynomial time.
NW-JSP	No-wait Job-shop Scheduling Problem.
NW-PSP	No-wait Packet Scheduling Problem.
OMNeT++	Objective Modular Network Test-bed in C++.

OT	operational technology.
PCP	Priority Code Poin.
RAM	Random-Access Memory.
RGG	random geometric graph.
SMT	Satisfiability Modulo Theory.
TAS	time-aware shaper.
TSN	Time sensitive networking.
UDP	User Datagram Protocol.
XML	Extensible Markup Language.

1 Introduction

In modern Cyber-Physical Systems (CPSs), real-time communication with guarantees on delay and jitter has become a key requirement for safety. Failure to meet certain real-time guarantees can result in significant economic or human damage in a variety of scenarios, like failure of air traffic control systems or autonomous driving systems. A real-time system is subject to real-time constraints, for example from event to system response. Real-time systems must ensure that responses are provided within bounded time constraints, which are often referred to as deadlines [Ben06]. For example, an airbag in a car has to unfold within 50 milliseconds in a crash [WG02]. Included in these 50 milliseconds is the time required for a sensor to detect the collision, the time necessary for the real-time system to transmit the signal to the airbag trigger mechanism, and the duration of the airbag inflation process. Furthermore, the timing of the airbag must be perfect so that it is inflated to its maximum as the occupant impacts onto it [WG02], because if it inflates too soon it is already deflating by the time the occupant hits the airbag and if it inflates too late the occupant will be in the inflation zone when the bag fires. Consequently, the real-time system is subject to a strict time constraint, which is only a fraction of the 50 milliseconds. As a result, real-time systems have evolved from a specialized field to a major area of research.

For a long time real-time systems were highly specialized to specific industries, like avionics, defense, and specific high-performance industrial applications [KS22]. In recent years, real-time systems have become increasingly prominent due to advancing automation. With the rise of the Internet of Things (IoT) the area of application for real-time systems also significantly increased [SSH+18]. In the context of the Internet of Things, real-time systems frequently necessitate the integration of IT components, which in turn facilitate the development of networked real-time systems. As the number of real-time systems that employ IT components continues to grow, the significance of networked real-time systems is becoming increasingly apparent. These networked real-time systems are fundamentally different from communication networks found in common IT infrastructures, as real-time systems must ensure timely message deliveries that are not required in IT networks. The key features of time-critical networked systems for the Internet of Things (IoT) and automotive industries are precision and safety. These features are achieved through deterministic transmissions. The deterministic transmission ensures that messages are delivered within a predictable time frame.

Originally network technologies for deterministic real-time systems were tailored to the specific real-time requirements for so-called operational technology (OT) [FGD+22]. In light of the growing importance of deterministic real-time systems and the rapid advancement of network technologies, like Ethernet, the Institute of Electrical and Electronics Engineers (IEEE) has merged Ethernet and deterministic real-time systems into Time-Sensitive Networking (TSN) [16]. This standard includes traffic shaping mechanisms for different real-time traffic and the time synchronization between the TSN switches. Although the IEEE standard for TSN establishes the technological and methodological infrastructure necessary for real-time network operations, it does not specify the schedule computation.

Computing an efficient and robust schedule within a TSN network is a (NP-)hard problem [DN16a], therefore the scheduling problem was originally addressed for static, a priori known sets of flows [FGD+22]. With the need for modern real-time systems to be flexible and able to quickly add and configure devices on the network, the concept of static environments is becoming increasingly untenable. This led to the development of a substantial body of research in the field of scalable schedulers for TSN networks [SOLM23].

The majority of the scheduling algorithms presented in these studies are theoretical and have not been subjected to testing or evaluation on many TSN networks, like in [GP18] the largest evaluated network has 18 bridges or in [HAD+21] were the networks only consists of two bridges with respectively three end devices. These scheduling algorithms require further testing and evaluation before deployment. This thesis presents an approach to the testing and evaluation of TSN scheduling algorithms, through simulation. We have developed and implemented a pipeline that transforms the computed schedule of an algorithm into a functional simulation of the schedule. In order to conduct the simulation, we use the discrete event simulator OMNeT++ [Var01] [VH10], which is then extended with the INET framework [MVK19]. The simulation provides researchers with a controlled, flexible, and cost-effective enviroment of testing and evaluating the scheduling algorithm prior to its deployment in the physical world. We have decided to restrict our simulation to the algorithm of [FGD+22], which is based on the scheduling algorithm of [FDR20], in the optimized [GDR] form, given that the various other schedulers have disparate output formats for the schedule. This algorithm employs a conflict-graph-approach to compute schedules for TSN networks.

Another contribution of this work is the evaluation of the simulated schedules as well as the performance of our pipeline for the simulation. The evaluation compares the simulated transmission and receiving times with the computed times out of the schedules. Furthermore, it serves to verify the zero-queuing methodology employed in the schedule's computation by [GDR]. Therefore the findings offer valuable insights regarding the viability of the schedule algorithm prior to its deployment on real TSN networks, thereby bridging the gap between theoretical and practical applications.

Chapter 2 presents an introduction to the necessary technical background to understand this study. Subsequently, Chapter 3 presents existing research on scheduling algorithms in TSN networks, as well as simulations of TSN networks in OMNeT++/INET. In Chapter 4 we explain our implementation of the pipeline. This chapter is divided into two sections. The first section presents the actual simulation in OMNeT++/INET , while the second section outlines the implementation of a pipeline in Python to transform the computed schedule of [GDR]. Following that, in Chapter 5, the schedule is analyzed and evaluated based on the outcomes of the simulation conducted in OMNeT++/INET. Finally, we conclude our work in Chapter 6 and offer suggestions for future improvements to our simulation.

2 Preliminaries

In this chapter, we provide an overview on Time-Sensitive Networks and explain some basic components and functions, namely the time-aware shaper and time synchronization. Further, conflict graphs are formalized, which we used to solve the traffic planning problem. Finally, we broadly introduce the network simulation environment OMNeT++/INET.

2.1 Time-Sensitive Networks (TSN)

Real-time systems must be able to transfer data deterministically, to be able to comply with their QoS guarantees, like bounded end-to-end delays and jitter. Therefore, network technologies with deterministic real-time guarantees have been available for quite some time. Popular examples include PROFIBUS [TV99] or EtherCAT [Pry08] [JB04]. Most of these technologies are based on Ethernet but incompatible with one another.

For this reason, the IEEE 802.1 Time-Sensitive Networking (TSN) Task Group of the Institute of Electrical and Electronics Engineers (IEEE) has specified standards for IEEE 802.3 networks enabling deterministic real-time communication over standard Ethernet [FHC+19] [16]. These standards include traffic shaping mechanisms for different real-time traffic and the time synchronization between the TSN switches, which will be discussed in the following paragraphs.

2.1.1 TSN Switch

The main components of TSN networks are switches, which are used to forward frames in the network. The functionality of a TSN switch, also called bridge, can be split up into three stages: ingress, message-switching, and egress [NTA+19].

During the ingress phase, incoming frames are classified by specific criteria like priority, source, and intended destination. This classification provides the basis for subsequent operations. In the second phase, the message-switching phase, the frame's path through the switch is determined. At this point, it is ensured that time-critical frames are given the highest priority and are not subject to unreasonable delays. The egress phase manages the final transmission of the frames. At this stage, advanced queue management techniques and priority mechanisms such as traffic shapers are employed to ensure schedules or traffic shapes are met. In this paper, we will focus on the time-aware shaper, which will be discussed in the next section.

2.1.2 Time-Aware Shaper

Within TSN there are different traffic shapers, like the time-aware shaper (TAS) or the credit-based shaper (CBS), which is mostly used for audio/video bridging [FHC+19]. In this paper, we will use the time-aware shaper because it allows precise per frame scheduling within TSN.

The Time-Sensitive Networking Task Group standardized the time-aware shaper (TAS) in IEEE Std 802.1Qbv [16]. The foundation of the IEEE Std 802.1Qbv is a time-triggered gating mechanism with port local priorities [FHC+19]. Each egress port of a switch has up to eight queues. Frames are guided to one of the queues through a dedicated header field, the so-called Priority Code Point (PCP). Gates control the transmission of buffered frames within a queue. A buffered frame can only be transmitted through an open gate. The state of each gate is determined in the Gate Control List (GCL), defining when gates open or close. To ensure determinism the GCL has to be strictly synchronized in time.

Whenever multiple queues have frames and open gates simultaneously, the frames are forwarded with strict priority scheduling based on their PCP [FHC+19]. When a gate closes during a transmission, the transmission continues. This can delay the transmission of the frames in queues with now open gates. To prevent these undesired delays, guard bands were introduced. A guard band denotes the time where the open gate is closed earlier so that any ongoing transmission is finished before a closed gate opens [DN16a]. This is relevant to prevent time-triggered real-time critical traffic from being delayed by best-effort traffic.

2.1.3 Time Synchronization

Time synchronization is one of the keystones of TSN. Without time synchronization the time-aware shaper would not be deterministic. This synchronization enables the transmitting and receiving of frames in precisely defined time frames. The IEEE 802.1AS standard [11] offers a robust framework for time synchronization within TSN. This standard uses the generalized Precision Time Protocol (gPTP, IEEE Std 1588) [20] to guarantee that the clocks of all switches are synchronized. In simulations, the IEEE 802.1AS standard demonstrated its high performance, maintaining time synchronization even with a large number of switches [TG08].

2.2 Conflict-graph-based Scheduling

Finding schedules in large TSN networks, where every transmission of a frame is spatially or temporally isolated, is an NP-hard problem. This spatial and temporal isolation is achieved by the selection of a route and offset (phase) for each stream so that their frames do not interfere with each other. To compute such a schedule a conflict-graph-based algorithm can be used [GDR].

A conflict graph is an undirected, vertex-colored graph $\mathcal{G}_c = (\mathcal{V}, \mathcal{E}, C)$. Here, $\mathcal{V}$ is the vertex set, $\mathcal{E} \subseteq V \times V$ the set of undirected edges V × V , and C the set of colors. Further, let the function $c(v) : \mathcal{V} \rightarrow C$ map vertices to their respective color.
To map the scheduling problem onto the graph, the vertices model possible scheduling configurations (combination of a candidate route and phase), and the vertex colors denote the streams [GDR].

So every stream has an unique color. Conflicts between two configurations are represented as edges between the configuration vertices. A conflict arises, whenever two configurations are neither spatially nor temporally separated, i.e., they are mutually exclusive.

To solve the scheduling problem an independent colorful set (ICS) is used [FGD+22]. An ICS is a refinement of the independent set (IS). The IS is a set of vertices in a graph $\mathcal{G}(\mathcal{V}, \mathcal{E})$ where no two of the vertices in the set are adjacent.

$$IS(\mathcal{G}) \subseteq \mathcal{V} | \forall v_a, v_b \in \mathcal{V}, a \neq b : (v_a, v_b) \notin \mathcal{E} \tag{2.1}$$

The ICS requires a vertex-colored graph $\mathcal{G}_c$ and then refines an IS so that each vertice in the set has a unique color and the size of the IS is maximized.

$$ICS(\mathcal{G}_c) = maxIS(\mathcal{G}_c) | \forall v_i, v_j \in IS, i \neq j : c(v_i) \neq c(v_j) \tag{2.2}$$

The best possible result is a set that contains every color ($|ICS| = |C|$), meaning in our mapping a schedule that admits every stream. However, it is not possible to find such an optimal set for every scheduling problem. Therefore, one often has to fall back to finding a good ICS, i.e., a set that contains most colors. Further, a maximal (or good) set can often be found in a substantially smaller subgraph of $\mathcal{G}_c$. Hence, the computation time for building the conflict graph and finding an ICS can be reduced by building $\mathcal{G}_c$ accordingly [GDR].

2.3 OMNeT++/INET

Simulation is a cost-effective way to validate the effectiveness of schedules. Therefore the discrete event simulator OMNeT++ (Objective Modular Network Test-bed in C++) is used [Var01] [VH10]. We further use the INET framework in OMNeT++ to simulate TSN networks [MVK19].

OMNeT++ is a public-source, generic simulator and supports the simulation of any networks with discrete events, like computer networks. It provides an Eclipse-based Integrated Development Environment (IDE) as well as a graphical user interface for the simulations. Headless simulations with output files for further use and evaluation are also possible. When modeling networks, OMNeT++ follows a hierarchical and modular approach, making it easy to reuse and configure components. These components can be nested to any depth allowing OMNeT++ to support the simulation of large and complex networks. It is also possible to run a simulation on the same device with different parameters. The parameters of the different components can be changed either in the associated OMNeT++ file or in special configuration files.

The INET framework extends OMNeT++ for communication networks [MVK19]. INET is an open-source project with many contributors and can freely be modified and used. It provides models for the internet stack, like UDP or IPv4, and its implementation follows the OSI layer architecture, providing link layer protocols like Ethernet. Within the Ethernet components, INET provides the implementation of TSN standards for switches and host devices.

3 Related Work

In this chapter, we take a look at the research that has already been done on traffic planning for TSN, the conflict-graph-based scheduling approach, and the simulation of TSN networks in OMNeT++/INET .

Traffic planning for TSN networks is a highly researched topic. A very common way to solve the scheduling problem is using mathematical solvers, like Integer Linear Programming (ILP) [VBHT22], [FDR18], Constraint Programming (CP) [VHT21], or Satisfiability Modulo Theory (SMT) [COCS16], [Ste10]. In [SCO18], the authors describe the TSN scheduling in general, focusing on the IEEE 802.1Qbv standard and the time-aware shaper. They explain scheduling at the switch level, allowing for refined control over frame-forwarding times. They further formalized a full reference model for TSN networks to understand how communication schedules are created. This formalization is also used to generalize the scheduling problem in TSN networks. SMT-solvers were used on small and medium network instances to synthesize schedules and verify existing ones. The experimental results show that varying the number of streams and windows affects the calculation time for a schedule drastically.

The traffic planning for TSN networks is also addressed in [AHM20], in which Atallah et al. present an approach revolving around the no-wait scheduling concept and iterated integer linear programming-based scheduling (IIS) techniques. The Degree of Conflict (DoC) between the IIS iterations is minimized by the DoC-aware streams partitioning (DASP) technique and the DoC-aware multipath routing technique guarantees the fault-tolerance. The authors also explore DoC-Aware Multipath Routing (DAMR), which generates optimized stream sets. The procedure builds an initial solution from these optimized sets and refines the solution using greedy heuristics that ensure efficient and fault-tolerant routing in TSN networks. In terms of performance, the DoC-aware iterative routing and scheduling (DA/IRS) method demonstrates remarkable scalability, speed, and success rate improvements (from 47% to 90%) over existing techniques such as ILP-based and pseudo-Boolean joint routing and scheduling, in networks with 21 bridges and 480 messages.

While the discussed methods of traffic planning in TSN networks are functional in small networks, it is not feasible to calculate an optimal solution for the scheduling problem with mathematical solvers in large-scale implementations. Those solver-based approaches are time-consuming and inefficient, which leads us to consider approaches that prioritize speed and scalability.

For better scalability of the solver-based approaches, heuristics are used. Dürr and Nayak modeled the scheduling problem, in [DN16b], as a No-wait Packet Scheduling Problem (NW-PSP) and mapped this onto a No-wait Job-shop Scheduling Problem (NW-JSP). The NW-JSP is making it possible to use an efficient tabu-search (meta-)heuristic on top of the ILP solver. This improves the scalability and efficiency of the ILP solver. The authors showed that their algorithm can compute near-optimal schedules for over 1500 flows in topologies with various sizes (24-100 hosts, 5-20 switches).

While this heuristic improved the scalability it is still limited by the use of a mathematical solver. For even better scalability approaches are needed that are not based on mathematical solvers. Such an algorithm was presented in [BAP+22] and is purely based on a heuristic. Unlike any other algorithm, the heuristic multi-queue scheduler (HERMES) decides the schedule link-by-link, starting with the destination link, working backward through the network, and ending with the source link. The link-by-link decision has the benefit to the frame-by-frame decision that conflicts between frames can be detected before the entire frame has been scheduled. In each link, each frame is scheduled as late as possible according to their time constraints, so that the preceding links have enough time margin between the frame's period start and the offset of the same frame in the last scheduled link. In tests, HERMES is hundreds of times faster than a normal CP scheduler. However, HERMES is limited because the routes in the network cannot present triple dependencies in a loop, those dependencies cause a deadlock in the algorithm as some links in the route will be indefinitely delayed.

While this approach is promising, unconstrained algorithms are needed to solve the scheduling problem in TSN networks for every occasion. In [FDR20] the authors present an approach to solve the traffic planning for time-triggered traffic in TSN networks with conflict graphs. The authors use a conflict graph to model the spatial and temporal isolation of the transmission of every stream. With this approach, it is sufficient to work with subgraphs to compute the schedule, making this approach even more efficient and scalable. To solve the scheduling problem an independent colorful set (ICS) is used. The ICS can be solved by efficient graph algorithms like Luby's algorithm. However, the conflict-graph-based approach is restricted as synchronization protocols like PTP are not included in the calculation. The evaluation of the proof-of-concept implementation showed that the conflict-graph-based approach is faster and more memory efficient compared to constraint-based approaches.

The conflict-graph-based scheduling approach was redefined in [FGD+22]. Falk et al. introduce an offensive planning approach, allowing for the reconfiguration of active flows for better resource utilization while providing QoS guarantees during such reconfigurations. To find colorful vertex sets in a conflict graph faster, the Greedy Flow Heap Heuristic (GFH) was introduced. This heuristic was specially designed to solve ICS problems quickly and efficiently. They demonstrated that the improved conflict graph approach handles hundreds of active flows in a dynamic scenario and it showed improvements to existing algorithms.

In [GDR], the conflict graph scheduling approach has been further refined. Previously the creation of the conflict graph was a significant bottleneck of the algorithm. Geppert et al. present multiple heuristics to create and expand the conflict graph, making the approach feasible for large-scale systems. With their novel conflict graph expansion techniques, they were able to schedule networks with hundreds of bridges and several thousand streams within minutes and perform updates in seconds.

[Asj24] emulated the conflict-graph-based scheduling approach in a virtual environment. The authors focused on the automatic creation of a virtual testbed, the deployment of the schedule, and the empirical evaluation of its performance. Within the evaluation, they found significant inaccuracies in emulating real-world hardware timing mechanisms in a virtual testbed. There are notable inconsistencies in frame drops and transmission accuracy, as the emulation is unable to replicate the precise timing and buffering requirements of real-world hardware.

Given the inaccuracies of the emulation, in this paper we validate the conflict-graph-based approach by simulating it. For this purpose, the already extensively researched OMNeT++ simulator is used. To facilitate the analysis of TSN networks, Falk et al. presented a TSN extension to the INET framework of the discrete event simulator OMNeT++ in [FHC+19]. This extension, called NeSTiNg, includes simulation model components for the time-triggered scheduling mechanism of IEEE Std 802.1Qbv like the time-aware and credit-based shapers. It also provides VLAN tagging, thus supporting strict-priority scheduling. Through a proof-of-concept implementation, NeSTiNg demonstrated its ability to simulate large TSN networks in a reasonable amount of time. The INET framework has since adopted NeSTiNg's TSN features.

In [Hau23], the author researched challenges regarding time-critical applications arising in wireless networks and provided possible solutions for these challenges. Important to our paper is the model for per-stream scheduling in a simulated TSN network in OMNeT++ that was also presented. To distinguish between different streams, a unique multicast address is assigned to every stream. Only the intended receiver of a stream is subscribed to its multicast address. This allows each stream to be routed independently.

4 Implementation

In this chapter, we explain the implementation of the simulation of the GFH algorithm. The implementation is split into two parts. The first part is the actual simulation of the approach in OMNeT++/INET . The second part is the implementation of a pipeline in Python to automatically transform a given schedule of the GFH scheduling algorithm of [GDR] into an OMNeT++/INET scenario.

For easier understanding, we explain both implementation parts based on a small example TSN network, shown in Figure 4.1. This network consists of two TSN devices and three TSN switches, which are connected via interfaces. In our example, two streams are sent from the end device A to the end device B. Stream A is sent to the end device B via switch 0 and switch 2, and stream B is sent to the end device B via switch 0, switch 1, and switch 2. So we have two streams with the same source and destination but different routes.

4.1 OMNeT++/INET

In this section, we explain how we have implemented our approaches in OMNeT++ using the example network (Figure 4.1) mentioned above. In the following, we will refer to OMNeT++/INET as the simulator. Since per-stream routing is required to simulate the conflict graph approach, we will first explain the implementation of the routing in the simulator, as it is not provided out of the box. We also explain the main configuration file of the simulation. Since synchronization protocols such as PTP are ignored in the conflict-graph-based approach, the simulation will also ignore these protocols.

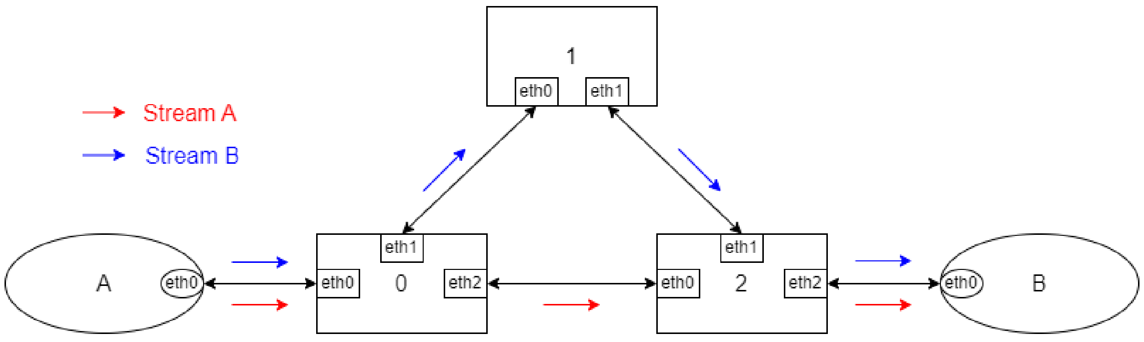

Figure 4.1: Example TSN network with two streams

Listing 4.1 XML file of the *Ipv4NetworkConfigurator* with two IP addresses for the same end device

```
<config>
  <interface hosts="*" address="172.x.x.x" netmask="255.x.x.x" />

  <multicast-group hosts="end_device_B" address="224.0.0.2" />
  <multicast-group hosts="end_device_B" address="224.0.0.3" />
</config>
```

vlanID	MAC address	outgoing interface
0	01-00-5E-00-00-02	eth1
0	01-00-5e-00-00-03	eth2

Table 4.1: MAC forwarding table of switch0 of the example

4.1.1 Routing

To simulate the GFH algorithm we need a model for per-stream routing. As in our example of Stream A and Stream B, we have different routes for streams with the same source and destination device. Therefore, we cannot route based on the IP or MAC address of the sender or receiver. In order to distinguish between the different streams, a unique multicast address is assigned to each of them. Only the intended receiver of a stream is subscribed to its multicast address. This model of per-stream routing in the simulator was reproduced from Haug et al. in [Hau23]. This allows each stream to be routed independently.

We use the manual IPv4 configurator (*Ipv4NetworkConfigurator*) provided by INET to assign multicast IP addresses to the different streams. To route stream A and stream B in our example network (Figure 4.1), we add two multicast groups to the IPv4 Network Configurator XML file (Listing 4.1). Separate end devices can have multiple multicast IP addresses as shown in the example in Listing 4.1. Each of the IP addresses in the XML file represents one stream.

OMNeT++/INET uses by default the shortest path to route streams/messages through the network. To achieve the per-stream routing we need to set the *MacForwardingTable* for every switch in the network. This module handles the mapping between ports and MAC addresses. The *MacForwardingTable* in the simulator can be pre-loaded from text files. Table Table 4.1 shows the structure of a *MacForwardingTable* for switch 0 from the running example. The first parameter of the Table is the vlanID, the second parameter is the MAC address of the multicast IP address (here 224.0.0.2 and 224.0.0.3), and the third parameter is the outgoing interface of the switch. Together the individual *MacForwardingTables* depict the custom routes. In our example at switch 0 frames from stream A are forwarded to switch 2 via Eth2 and frames from stream B are forwarded to switch 1 via Eth1, as also shown in Figure 4.1.

command	**value**	**explanation**
network	*TsnNetwork*	network to simulate
sim-time-limit	*1 μs*	simulation time in μs multiple of the hyper cycle of the schedule
***.bridging.directionReverser .delayer.typename*	*"PacketDelayer"*	enable packet delayer for processing delay on the switches
***.bridging.directionReverser .delayer.delay*	*4μs*	set packet delayer for processing delay on 4 μs
***.igmp.enabled*	*false*	IGMP messages allow devices to joins multicast groups. disabled since every TSN component in the simulated network knows its multicast groups from the Ipv4NetworkConfigurator
***.multicastForwarding*	*true*	forwarding of multicast messages in the components enabled to achive multicast in the network
***.defaultMulticastLoop*	*false*	self loop of multicast messages disabled because they are not needed and will crash the implementation
***.configurator.config*	*xmldoc("config.xml")*	XML file of the Ipv4NetworkConfigurator used to assign multicast IP adresses to the different streams
***.ethernet.macHeaderChecker .promiscuous*	*true*	true to process multicast messages at the end devices
***.app[*].io .joinLocalMulticastGroups*	*true*	enable that every UDP app joins local multicast groups

Table 4.2: Explanation of the general commands in the config file (*omnett.ini*)

4.1.2 Config

OMNeT++ is split into two main files, the *network definition file (NED)* and the *omnetpp.ini* file. In the *NED* file components are defined and assembled into networks. The *omnetpp.ini* file is the config of the simulation. It tells the simulation program which network to simulate (as NED files may contain several networks), or you can pass parameters to the simulation. In the following, we will explain our *omnetpp.ini* file to simulate the computed schedules of the GFH algorithm in Table 4.2 and Table 4.3.

4.2 Pipeline

In the previous section, we explained the configurations used to simulate the GFH algorithm [FGD+22] in OMNeT++/INET. To evaluate the GFH algorithm we want to simulate the schedules in OMNeT++/INET. To this end, we create these simulations automatically from the network files and corresponding graph-based schedules. The workflow of the automation pipeline will be discussed in the following.

command	**value**	**explanation**
applies to all switches in our example: switch 0 (**.switch0.)		
.macTable.forwardingTableFile	*"macTableSwitch0.txt"*	TXT file of the MacForwardingTable MacForwardingTable are preloaded from the TXT file
.hasEgressTrafficShaping	*true*	enable time aware traffic shaping
.bridging.directionReverser .reverser. excludeEncapsulationProtocols	*ïeee8021qctag"*	customer tag of IEEE 802.1ad allows for double VLAN tagging disabled because not needed in our simulation
applies to all end devices which are a source of a stream in our example: endDeviceA (**.endDeviceA.)		
.hasOutgoingStreams	*true*	enable outgoing streams for every TSN device that wants to send streams
.numApps	*int*	amount of UDP basics app on this source as every stream has its own app on his source node
applies to all apps of the end devices which are source of a stream (**.endDeviceA.app[0].)		
typename	*ÜdpSourceApp"*	application for the transmisson of streams
typename	*UdpSinkApp*	application for the receiving of streams
io.localport	*100*	local port
io.destPort	*100*	destination port
.source.packetLength	*1000 bytes*	frame size in bytes
.source.initialProductionOffset	*30 μs*	application start time (start of the first packet) in μs
.source.productionInterval	*34 μs*	sending period in μs
io.destAddress	*Ip adress*	ip adress of the destination
.io.interfaceTableModule	*"^.^.interfaceTable"*	enables multicast interface tables

Table 4.3: Explanation of the commands for the switches as well as the end devices in the config file (*omnett.ini*)

First, we summarize the output and functionalities of the GFH scheduling algorithm. Second, we lay out our implementation of the pipeline to get from the scheduler's output files to a working OMNeT++/INET project. This includes the generation of the *network definition file* (*NED*) in OMNeT++/INET from a given edge list of the network by the conflict-graph-based algorithm, as well as the generation of the configuration file (*omnetpp.ini*) and per-stream routing for our simulation form the computed schedule of the GFH algorithm.

4.2.1 GFH algorithm

To calculate a schedule the conflict-graph-based GFH algorithm [FGD+22] uses an edge list and a network traffic scenario. The edge list file represents the network topology and distinguishes between different types of nodes (device and switch nodes) within the network. The authors of the

Dynamic Flow Scheduler tool [FGD+22] provide a Python script to quickly generate edge lists for synthetic networks with given parameters, such as the number of switches and end devices. In our pipeline, we will transform these network topology in an OMNeT++/INET compatible format.

Another Python script from the Dynamic Flow Scheduler tool can be used to generate sample network traffic for a given network topology. This script makes use of a configuration file that contains the key parameters for generating different types of traffic scenarios. These parameters include the path to the edge list for the network topology, the number of initial streams, a list of possible frame sizes, and a list of possible frame transmission intervals. With this script, a wide range of traffic scenarios can be generated, which is important for the evaluation of our simulation of the conflict-graph-based algorithm.

With the network topology as an edge list and the network traffic scenario as a JSON file, the GFH algorithm can then compute a schedule and store it in a JSON file. This file includes the routing and scheduling information of every stream. We use it to derive the routing for OMNeT++/INET and to generate the configuration file.

4.2.2 NED file generation

The first step in the pipeline is to fill the *network definition file* (*NED*) in OMNeT++/INET, which stores the network topology. As explained in the previous section (Section 4.2.1) the GFH algorithm also needs the network structure to compute a schedule. In the algorithm, the network topology is given as an edge list. Our pipeline transforms the edge list into a *NED* file.

Firstly, it is necessary to sort the edge list into two categories: network devices and switches, because the *NED* file defines these network components. The file is sorted by identifying nodes with only one neighbor. This is because end devices in the present implementation of the GFH algorithm are required to be connected to only one switch. This condition is not a result of the system design of the GFH algorithm, but rather of the algorithm's implementation. The second step is to connect the network components and define the connection interfaces (eth) on adjacent components. The connection of the network components is also given in the edge list and is stored in a map. In this map, we also add to the connection the outgoing interface of each component of a connection. By iterating over this connection map we can define the connections with the outgoing interfaces between the network components in the *NED* file. This file also defines the default bitrate of every connection. In [GDR] a default bitrate of 1000 Mbps was used, as this is a common bitrate in modern networks. In order to simulate the computed schedule of the GFH algorithm, it is necessary to adopt this bitrate. We further set the propagation delay for every network link in this file at 1 μs.

Returning to our example (Figure 4.1), the *NED* file defines the TSN components, such as devices (end device A, end device B), switches (0-2), and the connections between them. The outgoing interfaces are also visible as eth for any connection between the nodes, such as switch 0 and switch 1 are connected through interface 1 on switch 0 and interface 0 on switch 1.

4.2.3 Configuration file generation

The second step of the pipeline is the generation of the *omnetpp.ini* file. This file contains the configuration parameters for the simulation, as illustrated in Table 4.2 and Table 4.3. The configuration can be divided into four distinct segments. The initial segment delineates the general parameters for establishing a TSN network in OMNeT++/INET, as shown in Table 4.2. The other segments are shown in Table 4.3. The second segment is dedicated to configuring the parameters of the network switches. The third segment establishes the general parameters of end devices, which function as the source of a stream. In the fourth segment, the parameters governing the actual transmission of streams are established. The aforementioned parameters are applicable to all UDP basic applications utilized for transmission in OMNeT++/INET, on end devices, which serve as the source of a stream. In the following, we will provide an explanation of the generation of this configuration.

The initial parameters of the configuration file are utilized to establish a TSN multicast network. Hence, those parameters are similar for each simulation. The initial parameter defines the network (*NED file*) to be simulated, which was discussed in Section 4.2.2. The simulation time limit is derived from the computed schedule of the conflict graph approach (Section 4.2.1). The schedule includes the duration of one hypercycle, which signifies that the network is simulated until it reaches back the initial state. We added a configuration to the pipeline, which allows the user to select the number of hypercycles to be simulated. We further add a packet delayer on each switch of the network, which delays data frame which travel through the switch by 4 μs. This delayer depicts the processing delay of each switch. The IMGP messages are disabled as they are not required for our multicast routing process, which is being constructed using the *IPv4NetworkConfigurator* and *MacForwardingTable* from OMNeT++/INET . We use the *IPv4NetworkConfigurator* to assign distinctive multicast IP addresses to each stream, using a XML file, while the *MacForwardingTable* is utilized to create bespoke MAC tables for each switch. The routing information utilized in the generation of the Mac tables is derived from the computed schedule of the conflict-graph-based approach. In order to work with the *Ipv4NetworkConfigurator*, we need to link the generated IP network XML file to the simulation. As our network is using multicast it is necessary to enable the forwarding of multicast messages in each component of the network. It is also necessary to disable the self-looping of multicast messages in our components, as it is not required and results in the simulation crashing. The promiscuous of the MacHeaderChecker has to be set to true as otherwise, end devices cannot process received multicast messages. The final parameter of the initial segment serve to ensure that each subsequent application generated within this file joins local multicast groups. These applications are used for the transmission and receiving of the streams.

The next segment of the *omnetpp.ini* file applies to each switch in the simulation. For routing through our network, we need to link the MAC forwarding table files to each corresponding switch (c.f. Section 4.2.4). We also need to enable egress traffic shaping to use the time-aware shaper and disable the customer tag of IEEE 802.1ad which allows for double VLAN tagging which is not needed in our simulation.

The remaining configuration parameters apply to all end devices in the network that are sources or destinations of streams within the computed schedule of the GFH algorithm (Section 4.2.1). For end devices that serve as the source of a stream, it is necessary to enable them to have outgoing streams. The transmission and receiving of streams is facilitated by the utilization of dedicated applications within OMNeT++/INET. For the transmission of streams, we use a *UDPSourceApp* and

Algorithm 4.1 Pseudo code to generate the *MacTableFile*

```
procedure GENERATEMACTABLEFILE(streams, connectionMap)
    for all stream ∈ streams do
        macAdress ← IP2MAC($stream.ipaddress$)
        for all currentNode, nextNode ∈ PAIRWISE($stream.route$) do
            if currentNode.isSwitch then
                eth = connectionMap[currentNode][nextNode]
                CURRENTNODE.WRITETOMACTABLEFILE($macAdress, eth$)
            end if
        end for
    end for
end procedure
```

for the receiving a *UDPSinkApp*. In order to simulate the conflict graph approach, it is necessary to implement a per-stream routing strategy, as discussed in Section 4.1.1. This necessitates the generation of a transmission and receiving application for each stream. The applications are generated at the source device and at the respective end device of the stream. It is necessary for OMNeT++/INET to be aware of the number of applications generated at each end device. The parameters of each application, which represents a stream, are configured in the subsequent segment.

The last segment of the configuration defines the parameters of the actual UDP applications for each stream. These parameters are also derived from the computed schedule of the GFH algorithm (Section 4.2.1). The first parameter decides whether the application is a SourceApp (transmission) or a SinkApp (receiving). For the SourceApp the parameters include the start time of the transmission of the first packet of the stream as well as the period of the transmission. In order to facilitate the per-stream routing process, a multicast IP address is assigned to each individual stream, as detailed in reference Section 4.1.1. This IP address gets assigned to the destination address of the corresponding application of the stream in OMNeT++/INET. The local port and the destination port of the transmission are also specified. We also need to enable multicast interface tables for the end device. The last parameter is the message length, for our pipeline we need to reduce the given message length from the schedule by 66 bytes because OMNeT++/INET adds a header of 54 bytes and the Interframe gap with 12 bytes to each packet. This header is already included in the message length of the schedule. For the SinkApp the parameters include the local port, which has to be the destination port of the associated SourceApp, the multicast address of the stream, and the interface table module which enables multicast interfaces for the end device.

4.2.4 Routing

The routing in OMNeT++/INET had to be split into two parts, assigning multicast IP addresses for each stream and computing the *MacForwardingTable* for each switch, as described in Section 4.1.1. The partition as mentioned above was retained for the pipeline.

In the first part, we generate a unique multicast IP address for each stream that is given in the schedule of the GFH algorithm (cf. Section 4.2.1). In order to associate the destination of the stream with its corresponding multicast group, we store the generated multicast IP address in conjunction with the destination of each stream in an XML file, as illustrated in Listing 4.1. This file is used by the *Ipv4NetworkConfigurator* in OMNeT++/INET to assign the multicast IP address to each stream, as previously explained in Section 4.1.1.

In the second part, we compute and set the *MacForwardingTable* for each switch to route streams independently. To set the *MacForwardingTable* we need the MAC address of the multicast group of each stream that passes through the switch of the MAC table and the outgoing interface to the next network component on the stream's route, as shown in Table 4.1. This MAC address can be derived from the multicast IP address, as it identifies the receivers of each multicast group. In the context of our simulation, the sole receiver is the intended destination of the corresponding stream, as this was previously configured with the *Ipv4NetworkConfigurator*.

The pseudo-code to generate the *MacTableFile* is shown in Algorithm 4.1. We iterate through each stream and calculate the MAC address out of the previously set multicast IP address of the stream. The subsequent phase of the process requires the identification of the route traversed by the stream through the network. This information is provided in the computed schedule of the conflict-graph-based algorithm, as detailed in Section 4.2.1. We iterate through the route of the stream. If the current node is a switch, an entry is added to its MAC table with the computed MAC address and the outgoing interface to the subsequent node. This outgoing interface is determined by a connection map that is generated as previously described in the network generation process.

5 Evaluation

In the following, we describe our evaluation setup, followed by the metrics used to evaluate the scheduling algorithm of [GDR] with our simulation. Subsequently, we present the findings of our evaluation, commencing with the verification of the examined GFH scheduling algorithm of [GDR]. Following that we present the limitations and characteristics of the current implementation of the scheduling algorithm. We are completing our evaluation with the analysis of the performance of our implemented pipeline which transforms the computed schedules of the scheduling algorithm into a working simulation. This simulation is then executed in OMNeT++/INET.

5.1 Evaluation setup

Our evaluations are conducted on a server running Ubuntu 22.04.4. This server is equipped with two AMD EPYC 7413 24-core processors and a total of 256 gigabytes of RAM. For the implementation of the pipeline, we used Python 3.10.12. The implementation of our pipeline does not employ parallel programming. Furthermore, the OMNeT++ kernel is single threaded.

For a comparison with the results of [GDR], we evaluate the same network topologies, namely Waxman and Grid [RPGS17] [Wax88]. In the present implementation of the conflict-graph-based algorithm, these network topologies are formed by bridges, with every bridge being connected to one end device. For illustrative purposes, examples of the topologies can be found in Figure 5.1.

- **Grid**
  The grid topology arranges nodes in a square mesh. Every node is connected to its two to four closest neighbors, with the number of neighbors being two in the corners, three at the rim, and four inside the grid.
- **Waxman**
  In graph theory, the Waxman topology is a random geometric graph (RGG). A RGG is an undirected graph in which nodes are randomly distributed within a metric space. Two nodes are connected if the distance between them is under a specific parameter r. A Waxman topology introduces a probabilistic connection function for r, based on the Euclidean distance between the two nodes, favoring short distances. It can be argued that topologies that favor links to geographically proximate neighbors are more realistic, given that cables typically follow short paths between endpoints.

The network scenarios, generated by [GDR], are constructed by randomly selecting source and destination devices for each stream. Period and frame size are selected at random from a fixed set of values. The fixed values for frame sizes range from 125 to 1500 B, thus allowing for the transmission

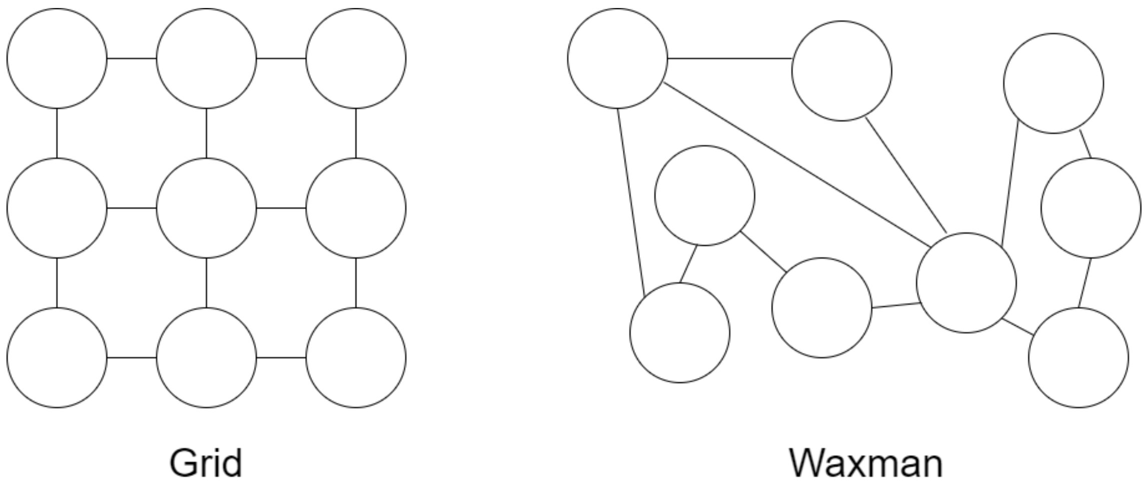

Figure 5.1: Example network topologies for Grid and Waxman, nodes shown in the figure represent bridges, at every bridge an end device is connected which is not represented in this figure

of both small messages and messages of the size approaching the maximum transmission unit (MTU). The periods in question range from hundreds of microseconds to a few milliseconds, and are selected to be harmonic, a common occurrence in numerous industrial applications [MNX+18].

The network speed in our simulation is 1 Gbit s^{-1} per network link with a propagation delay of 1 μs. Bridges within the network have a processing delay of 4μs [DK+14]. The network speed is derived from the same parameters as those used in the evaluation of [GDR] for comparison purposes. In our evaluation, we consider network sizes from 9 up to 49 bridges. The networks subjected to evaluation included the 3x3 Grid, the 7x7 Grid, and the 49-bridge Waxman topologies. For a given network size we evaluated 10 network instances for the Waxman topologies to avoid bias to a specific instance. For the Grid topologies, we only evaluated one instance per network size. For each topology and network size, we evaluated at least five different scenarios, each with at least five schedules. This means we evaluated at least 25 schedules per topology. The number of streams created per scenario varied depending on the network size. In order to ensure sufficient statistical power, 300 streams were created for each scenario in the 3x3 grid topologies. Similarly, 500 streams were created for the 7x7 grid, and 100 streams were created for the 49 bridges large Waxman topologies.

5.2 Performance Metrics

In this section, we discuss the metrics to evaluate the performance of the computed schedule by [GDR] as well as the performance of our pipeline.

- **Transmission time**
  This metric compares the computed transmission times of the scheduling algorithm with the transmission times in the simulation.

network topology	amount of scheduled streams
example network	5 streams
3x3 Grid	300 streams
7x7 Grid	500 streams
49 Waxman	100 streams

Table 5.1: Amount of scheduled streams per network topology

- **Receiving time**
  This metric compares the expected receiving times of the scheduling algorithm with the receiving times in the simulation. It indicates if the computed schedule is implementable.
- **End-to-End Delay** This metric measures the time a data frame takes to travel from its source device to its end device. Included in this metric are all network-related delays, like propagation, processing and transmission delay.
- **Max queue length**
  This metric measures the maximum amount of packets that are in a specific queue at the same time. It is necessary to verify that this metric is not greater than one, given that the scheduling algorithm described in [GDR] employs a zero-queuing approach.
- **Total number of packets**
  This metric measures the total number of packets that traveled through the network
- **Pipeline time**
  This metric measures the time our pipeline needs to transform a computed schedule into a working simulation

5.3 Verification of the scheduling algorithm

One of the primary objectives of this study was to verify TSN scheduling algorithms through simulation, with the scheduling algorithm proposed by [GDR] serving as a case study. We verify the scheduling algorithm through the presented metrics. We are starting with a comparison of the computed *transmission times* for each stream with the *transmission times* in our simulation. Next, we compare the *receiving time* of each stream, therefore we compare the computed *End-to-End delay* of the algorithm with the actual *End-to-End delay* in the simulation. Then the *maximum length* of a given queue at any given moment is also measured, to verify the zero-queuing approach to the scheduling algorithm, before we summarize and validate the computed schedules of the scheduling algorithms by [GDR].

5.3.1 Transmission time

The *transmission time* is a metric that compares the transmission times at the source device of each stream of the simulation with the computed schedule of the scheduling algorithm. The schedule delineates the transmission time into two distinct segments: the transmission time of the initial data frame of the scheduled stream and the period of the stream's transmission. For our simulation, we derive the transmission start time as well as the period of the transmission from the computed schedule.

The *transmission times* between the simulation and the scheduling algorithm match. This is because the transmission times are taken as parameters in the simulation from the schedule. This means there is no network traffic delaying the transmission of the next data frame in the computed schedule.

5.3.2 End-to-End Delays

Given that the transmission times between the algorithm and the simulation are aligned, it is sufficient to examine the *End-to-End delays* for the evaluation of the receiving times. The *End-to-End delay* measures the time a data frame of a stream takes to travel from its source device to its destination device. Included in this metric are all network-related delays, like propagation, processing, and transmission delay. To compare the simulation results with the scheduling algorithm we adopted the same values for these network delays as presented in the evaluation of [GDR]. This means we simulated networks with 1 Gbit s^{-1} network links with a propagation delay of 1 μs. The processing delay of the bridges was 4μs.

In our simulation, the *End-to-End delays* remain consistent within the different data frames of a stream. This consistency is independent of the different network topologies and network scenarios simulated. The consistency of the delays remains unaltered by the simulation of more than one hypercycle. The *End-to-End delays* of our simulation furthermore match the computed *End-to-End delays* of the scheduling algorithm in all the performed simulations. The *End-to-End delays* of the streams for the scheduling algorithm were calculated by adding the network delays for each stream.

5.3.3 Zero-queueing approach

The scheduling algorithm of [GDR] pursues a zero-queuing approach. This implies that only a single packet is present within a given queue at the same time. In our simulation, we measured this with the *maximum queue length.* To verify the zero-queuing approach of the scheduling algorithm this metric has to be zero or one, with zero meaning that no data frame was routed through the specific egress port of the queue. With our simulation, we can verify the zero-queuing approach of the scheduling algorithm. We can further verify that neither the network topology nor the network scenario has an impact on the *maximum queue length.* The *maximum queue length* is also unaltered by the simulation of several hypercycles of a schedule.

5.3.4 Validation of the schedule

As the measured metrics of our simulation match the expected values of the scheduling algorithm we have very good evidence that the computed schedules of the scheduling algorithm presented in [GDR] function. The transmission times and end-to-end delays are consistent and match the anticipated values, indicating that the computed schedule of the algorithm does not generate additional network traffic that could potentially disrupt the schedule. Furthermore, the scheduling algorithm is designed in accordance with the assumed zero-queuing principle.

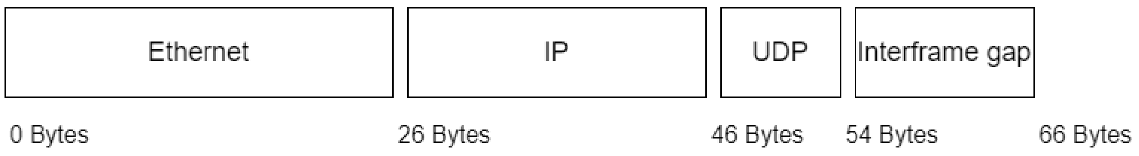

Figure 5.2: Breakdown of the 66 Bytes which need to be reduced by the simulation for the scheduled frame sizes

5.4 Limitations of the scheduling algorithm

In this section, we explain the limitations of the existing implementation of the scheduling algorithm presented in [GDR], commencing with the assumption of payload utilized by the scheduling algorithm. Afterward, we explain the assumptions of the algorithm for the network delays.

5.4.1 Frame sizes

In the present implementation of the GFH scheduling algorithm, the frame sizes indicated in the schedule correspond to the actual frame sizes that are transmitted within the network, rather than the payload. This means the payload in the schedule includes the Interframe gap as well as Ethernet and IP headers. For our simulation, we reduced the frame size of the schedule by 66 bytes. Figure 5.2 illustrates the distribution of the 66 bytes in question, as they include 26 Bytes for the Ethernet protocol, 20 bytes for the IPv4 protocol, 8 Bytes for the UDP protocol, and 12 Bytes for the Interframe gap. Failure to comply with this frame size limitation leads to fragmentation of the transmitted frames during the simulation and therefore to more network traffic as the scheduling algorithm expects. Hence the computed schedule of the algorithm can not be adhered to, if the frame sizes are not reduced.

5.4.2 Delays

For the network delays, the current implementation of the scheduling algorithm of [GDR] assumes homogeneous networks with fixed values. These delays include the processing delay at every switch, with 4 μs, as well as the propagation delay for each network link with 1 μs. For network speed, the current implementation of the scheduling algorithm assumes 1 Gbit s^{-1} network links, this results in a transmission delay per packet.

$$(5.1)\quad TransmissionDelay = \frac{FrameSize * 8}{1000}\mu s$$

Before deployment of the scheduling algorithm, these network delays need to be adjusted to the respective speed of the network links with the exact propagation delay and the processing delay of each switch.

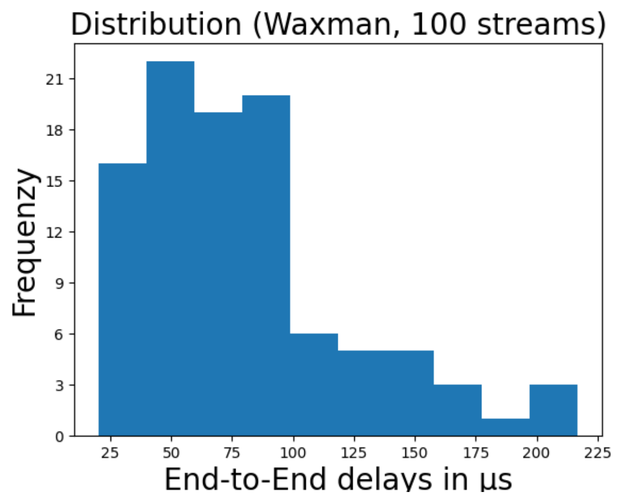

Figure 5.3: Distribution of the End-to-End Delay for 49 switches and end devices each, Waxman topology with 100 scheduled streams

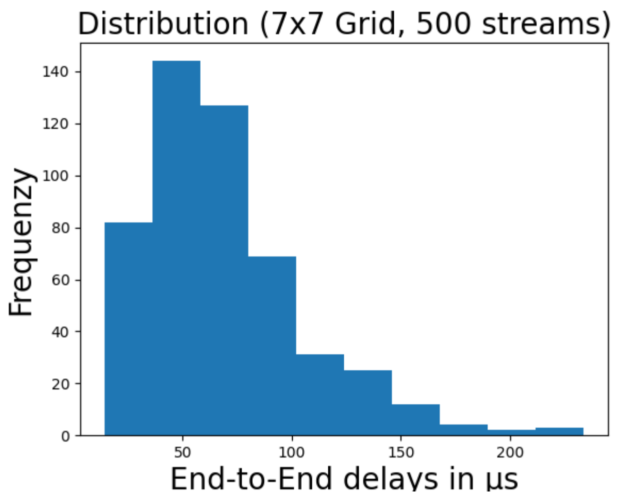

Figure 5.4: Distribution of the End-to-End Delay for 49 switches and end devices each, Grid topology with 500 scheduled streams

5.5 Characteristics of the scheduling algorithm

In this section, we analyze the characteristics of the scheduling algorithm. First, we show the distribution of the End-to-End delays for a Waxman topology with 100 streams and a 7x7 Grid topology. Afterward, we show the number of packets that are sent during different network scenarios and for different network topologies and sizes.

network topology	amount of scheduled streams	total number of packets
example network	5 streams	30 packets
3x3 Grid	300 streams	2067 packets
7x7 Grid	500 streams	3812 packets
49 Waxman	100 streams	732 packets

Table 5.2: Total amount of transferred packets for the simulation of one hypercycle

5.5.1 Distribution of the End-to-End delays

Figure 5.3 shows the distribution of the End-to-End Delays for a Waxman topology with 49 bridges and a network scenario of 100 scheduled steams. The End-to-End delay for most of the scheduled streams is below 100 μs, with a few exceptions which take up to 200 μs. Figure 5.4 shows that in a grid topology with 49 bridges and a network scenario with 500 scheduled steams, the shorter End-to-End delays are also favored. Long End-to-End delays have a decreasing frequency for this topology, with the longest End-to-End delay being 234 μs, which represents the diameter, so the length of the shortest path between the most distanced nodes, of the 7x7 topology and a packet size of 1500 bytes. This diameter is 14 for the 7x7 Grid topology. The bulk part of the End-to-End delays in this Grid topology takes less than 100 μs. This means that the distribution of End-to-End delays between the two topologies is very similar, despite the scheduling of 500 streams in the Grid compared to the 100 streams in the Waxman topology.

5.5.2 Total number of packets

We recorded the number of packets transferred over the network for the evaluated network topologies and scenarios of [GDR] in one hypercycle. These numbers of packets can be seen in Table 5.2.

The example network (Figure 4.1) with 5 streams transfers 30 packets in one hypercycle. A 3x3 Grid Topology with 300 streams transfers 2067 packets over the network, a 7x7 Grid with 500 streams 3812 packets, and a 49 Waxman topology with 100 streams 732 packets. This means for our evaluated topologies and scenario the amount of packets transferred is between fivefold and eightfold of the amount of scheduled streams. As the number of streams increases, the multiplication for the number of transferred packets also increases. It can be reasonably deduced that as the number of streams increases, the simulation time will also increase significantly, due to the fact that a greater number of packets will be transferred over the network.

5.6 Pipeline performance

In the following, we are going to evaluate the performance of our pipeline. In order to do this, we recorded the computational time to transform the output of the conflict graph-based algorithm to a working simulation in OMNeT++/INET, and the execution time to simulate the schedule. Those two times got added into the *pipeline time*. We evaluate the *pipeline time* for a variety of network sizes and scenarios. For every network topology, we evaluated at least five different scenarios with five schedules. The *pipeline time* per topology represents the mean time it took to transform the

network topology	amount of scheduled streams	pipeline time
example network	5 streams	2.09s
3x3 Grid	100 streams	3.32s
3x3 Grid	300 streams	9.12s
3x3 Grid	500 streams	89.43s
7x7 Grid	100 streams	20.82s
7x7 Grid	300 streams	39.12s
7x7 Grid	500 streams	386.46s
49 Waxman	100 streams	25,31s

Table 5.3: Average pipeline times for different network topologies and number of streams

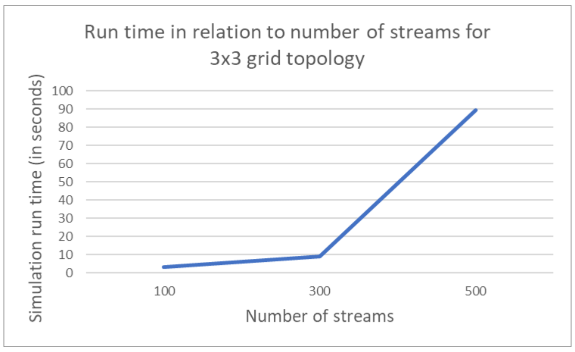

Figure 5.5: Runtime of the pipeline in relation to number of streams for 3x3 Grid topology

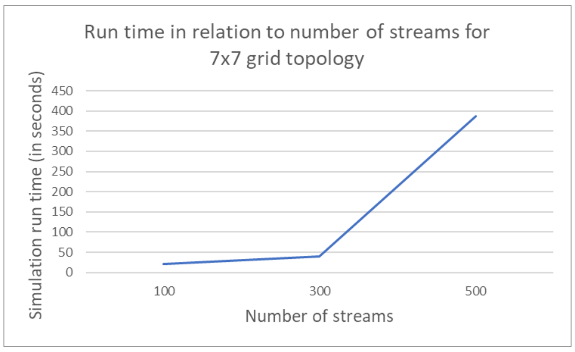

Figure 5.6: Runtime of the pipeline in relation to number of streams for 7x7 Grid topology

schedule and execute the simulation in our experiments. For the *pipeline time* we always simulated one hypercycle of the computed schedule by [GDR]. The *pipeline times* for the different topologies can be seen in Table 5.3.

The first topology we tested was our example network (Figure 4.1) and a scenario of five streams. To transform and execute the simulation of the example network (Figure 4.1) with a schedule of five streams our pipeline takes 2.09s. This time gets drastically better if the output directory of the simulation already exists with a time of 1.28s. For the 3x3 grid topology with a scenario of 100 streams, the pipeline takes an average of 3.32s. Here the pipeline time differs from 2.81 up to 4.07s between the different scenarios. The same network topology with 300 scheduled streams takes in the mean 9.12s and for 500 scheduled streams 89.43s. the drastic enlargement of the *pipeline time* as the number of streams increases can be seen in Figure 5.5.

For the 7x7 Grid topology, we started our evaluation of the *pieline time* also with a scenario of 100 scheduled streams. Our pipeline averages 20.82s for this topology and scenario. For a scenario of 300 streams on the same network topology, the pipeline takes in the mean 39.12s, and for 500 streams 386.46s. Again, the pipeline time increases dramatically as the number of streams increases, as shown in Figure 5.6.

We only evaluated the Waxman topology for a network size of 49 switches and 49 end devices, as a smaller Waxman topology is very similar to a fully randomized graph topology. The pipeline takes on avrage 25.31s for the transformation of the output of the scheduling algorithm into a working simulation and the execution of this simulation for the Waxman topology. For the different Waxman topologies evaluated, the *pipeline time* varied from 18.27s to 37.34s.

6 Conclusion and Outlook

In this thesis, we have investigated the problem of verifying scheduling algorithms for TSN networks, since many introduced scheduling algorithms are theoretical scientific studies. We proposed a simulation approach to verify and evaluate these scheduling algorithms. In order to demonstrate our approach, we have implemented a Python pipeline that takes the output of the scheduling algorithm and transforms it into a functional simulation within the OMNeT++/INET environment. As the different scheduling algorithms have different output formats we restricted this study to the verification and evaluation of the GFH scheduling algorithm of [FGD+22], which is based on the scheduling algorithm presented in [FDR20], in the optimized [GDR] form. This scheduling algorithm utilizes a conflict-graph-approach to compute schedules for TSN networks.

In our evaluation, we compared the simulation results with the schedule for different network topologies and network scenarios. The simulation results show that the evaluated schedules of the GFH scheduling algorithm work correctly for the respective network topology and network scenario. We can verify that the transmission times of packets between the simulation and the computed schedule match. For the receiving times of the packets, we compared the End-to-End Delay of the packets between the schedule and the simulation. The End-to-End delays in our simulation are consistent amongst packets of their specific stream, as expected by the scheduling algorithm. The delays further match between the simulation and the scheduling algorithm. For the zero-queueing approach of the scheduling algorithm, we measured the maximum length of the different queues at the egress ports of each switch at any given time. We verified that at maximum a single packet is present within a given queue at the same time. We further explained the limitations of the current implementation of the scheduling algorithm, presented in [GDR]. The implementation of the scheduling algorithm assumes homogeneous networks with fixed values for the network delays, like processing delays of the different switches, as well as network speed and propagation delays of the network links. Further, the frame sizes specified in the schedule correspond to the actual frame sizes that are transmitted on the network. This means the frame sizes of the schedule include all the data generated by Ethernet, UDP, and IP headers, as well as the Interframe gap. In our simulation, we therefore had to reduce the frame size of the schedule by 66 bytes. These fixed values for the frame sizes and network delays represent an implementation detail of the GFH algorithm, that can be resolved by acquiring comprehensive knowledge of the underlying TSN network and incorporating this information into the schedule computation. We also evaluated the time our pipeline needs to transform the computed schedule into a working simulation and the execution of this simulation. This pipeline time increases drastically with the number of scheduled streams per network scenario.

Outlook

Future work can focus on the simulation of other TSN scheduling algorithms with another approach to the conflict graph approach, to evaluate and compare the scheduling algorithms. Our simulation pipeline can be improved by implementing a multi-threaded version of the pipeline. This multi-threaded version would speed up the simulation of a large number of schedules per network topology and network scenario as this is needed to gain statistical significance of the simulated data. The implementation of this multi-threaded version would only affect the pipeline, given that the OMNeT++ kernel is single-threaded. However, it would facilitate the acceleration of the simulation of a substantial number of schedules through the concurrent execution of multiple simulations.

Bibliography

[11] "IEEE Standard for Local and Metropolitan Area Networks - Timing and Synchronization for Time-Sensitive Applications in Bridged Local Area Networks". In: *IEEE Std 802.1AS-2011* (2011), pp. 1–292. DOI: 10.1109/IEEESTD.2011.5741898 (cit. on p. 20).

[16] "IEEE Standard for Local and metropolitan area networks – Bridges and Bridged Networks - Amendment 25: Enhancements for Scheduled Traffic". In: *IEEE Std 802.1Qbv-2015 (Amendment to IEEE Std 802.1Q-2014 as amended by IEEE Std 802.1Qca-2015, IEEE Std 802.1Qcd-2015, and IEEE Std 802.1Q-2014/Cor 1-2015)* (2016), pp. 1–57. DOI: 10.1109/IEEESTD.2016.8613095 (cit. on pp. 17, 19, 20).

[20] "IEEE Standard for a Precision Clock Synchronization Protocol for Networked Measurement and Control Systems". In: *IEEE Std 1588-2019 (Revision ofIEEE Std 1588-2008)* (2020), pp. 1–499. DOI: 10.1109/IEEESTD.2020.9120376 (cit. on p. 20).

[AHM20] A. A. Atallah, G. B. Hamad, O. A. Mohamed. "Routing and Scheduling of Time-Triggered Traffic in Time-Sensitive Networks". In: *IEEE Transactions on Industrial Informatics* 16.7 (2020), pp. 4525–4534. DOI: 10.1109/TII.2019.2950887 (cit. on p. 23).

[Asj24] M. Asjadulla. "Deployment and Empirical Verification of Real-time Schedules". In: (2024). DOI: 10.18419/opus-13906 (cit. on p. 24).

[BAP+22] D. Bujosa, M. Ashjaei, A. V. Papadopoulos, T. Nolte, J. Proenza. "HERMES: Heuristic multi-queue scheduler for TSN time-triggered traffic with zero reception jitter capabilities". In: *Proceedings of the 30th International Conference on Real-Time Networks and Systems*. 2022, pp. 70–80 (cit. on p. 24).

[Ben06] M. Ben-Ari. *Principles of concurrent and distributed programming*. Pearson Education, 2006 (cit. on p. 17).

[COCS16] S. S. Craciunas, R. S. Oliver, M. Chmelík, W. Steiner. "Scheduling real-time communication in IEEE 802.1 Qbv time sensitive networks". In: *Proceedings of the 24th International Conference on Real-Time Networks and Systems*. 2016, pp. 183–192 (cit. on p. 23).

[DK+14] F. Dürr, T. Kohler, et al. "Comparing the forwarding latency of openflow hardware and software switches". In: *Fakultät Informatik, Elektrotechnik Informationstechnik, Univ. Stuttgart, Stuttgart, Germany, Tech. Rep. TR* 4 (2014), p. 2014 (cit. on p. 36).

[DN16a] F. Dürr, N. G. Nayak. "No-wait packet scheduling for IEEE time-sensitive networks (TSN)". In: *Proceedings of the 24th International Conference on Real-Time Networks and Systems*. 2016, pp. 203–212 (cit. on pp. 18, 20).

[DN16b] F. Dürr, N. G. Nayak. “No-wait packet scheduling for IEEE time-sensitive networks (TSN)”. In: *Proceedings of the 24th international conference on real-time networks and systems*. 2016, pp. 203–212 (cit. on p. 23).

[FDR18] J. Falk, F. Dürr, K. Rothermel. “Exploring practical limitations of joint routing and scheduling for TSN with ILP”. In: *2018 IEEE 24th International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*. IEEE. 2018, pp. 136–146 (cit. on p. 23).

[FDR20] J. Falk, F. Dürr, K. Rothermel. “Time-Triggered Traffic Planning for Data Networks with Conflict Graphs”. In: *2020 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*. 2020, pp. 124–136. DOI: 10.1109/RTAS48715.2020.00-12 (cit. on pp. 18, 24, 45).

[FGD+22] J. Falk, H. Geppert, F. Dürr, S. Bhowmik, K. Rothermel. “Dynamic QoS-aware traffic planning for time-triggered flows in the real-time data plane”. In: *IEEE Transactions on Network and Service Management* 19.2 (2022), pp. 1807–1825 (cit. on pp. 17, 18, 21, 24, 29–31, 45).

[FHC+19] J. Falk, D. Hellmanns, B. Carabelli, N. Nayak, F. Dürr, S. Kehrer, K. Rothermel. “NeSTiNg: Simulating IEEE Time-sensitive Networking (TSN) in OMNeT++”. In: *2019 International Conference on Networked Systems (NetSys)*. 2019, pp. 1–8. DOI: 10.1109/NetSys.2019.8854500 (cit. on pp. 19, 20, 25).

[GDR] H. Geppert, F. Dürr, K. Rothermel. “Efficient Conflict Graph Generation for Time-Sensitive Networks with Dynamically Changing Communication Patterns”. In: () (cit. on pp. 18, 20, 21, 24, 27, 31, 35–39, 41, 43, 45).

[GP18] V. Gavriluţ, P. Pop. “Scheduling in time sensitive networks (TSN) for mixed-criticality industrial applications”. In: *2018 14th IEEE International Workshop on Factory Communication Systems (WFCS)*. IEEE. 2018, pp. 1–4 (cit. on p. 18).

[HAD+21] B. Houtan, M. Ashjaei, M. Daneshtalab, M. Sjödin, S. Mubeen. “Synthesising schedules to improve QoS of best-effort traffic in TSN networks”. In: *Proceedings of the 29th International Conference on Real-Time Networks and Systems*. 2021, pp. 68–77 (cit. on p. 18).

[Hau23] L. Haug. “Roaming with deterministic real-time guarantees in wireless Time-Sensitive Networking”. In: (2023). DOI: 10.18419/opus-13092 (cit. on pp. 25, 28).

[JB04] D. Jansen, H. Buttner. “Real-time Ethernet: the EtherCAT solution”. In: *Computing and Control Engineering* 15.1 (2004), pp. 16–21 (cit. on p. 19).

[KS22] H. Kopetz, W. Steiner. “Real-time communication”. In: *Real-time systems: Design principles for distributed embedded applications*. Springer, 2022, pp. 177–200 (cit. on p. 17).

[MNX+18] M. Mohaqeqi, M. Nasri, Y. Xu, A. Cervin, K.-E. Årzén. “Optimal harmonic period assignment: complexity results and approximation algorithms”. In: *Real-Time Systems* 54 (2018), pp. 830–860 (cit. on p. 36).

[MVK19] L. Mészáros, A. Varga, M. Kirsche. “Inet framework”. In: *Recent Advances in Network Simulation: The OMNeT++ Environment and its Ecosystem* (2019), pp. 55–106 (cit. on pp. 18, 21).

[NTA+19] A. Nasrallah, A. S. Thyagaturu, Z. Alharbi, C. Wang, X. Shao, M. Reisslein, H. Elbakoury. “Performance Comparison of IEEE 802.1 TSN Time Aware Shaper (TAS) and Asynchronous Traffic Shaper (ATS)”. In: *IEEE Access* 7 (2019), pp. 44165–44181. DOI: 10.1109/ACCESS.2019.2908613 (cit. on p. 19).

[Pry08] G. Prytz. “A performance analysis of EtherCAT and PROFINET IRT”. In: *2008 IEEE International Conference on Emerging Technologies and Factory Automation*. IEEE. 2008, pp. 408–415 (cit. on p. 19).

[RPGS17] M. L. Raagaard, P. Pop, M. Gutiérrez, W. Steiner. “Runtime reconfiguration of time-sensitive networking (TSN) schedules for fog computing”. In: *2017 IEEE Fog World Congress (FWC)*. IEEE. 2017, pp. 1–6 (cit. on p. 35).

[SCO18] W. Steiner, S. S. Craciunas, R. S. Oliver. “Traffic planning for time-sensitive communication”. In: *IEEE Communications Standards Magazine* 2.2 (2018), pp. 42–47 (cit. on p. 23).

[SOLM23] T. Stüber, L. Osswald, S. Lindner, M. Menth. “A survey of scheduling algorithms for the time-aware shaper in time-sensitive networking (TSN)”. In: *IEEE Access* 11 (2023), pp. 61192–61233 (cit. on p. 18).

[SSH+18] E. Sisinni, A. Saifullah, S. Han, U. Jennehag, M. Gidlund. “Industrial internet of things: Challenges, opportunities, and directions”. In: *IEEE transactions on industrial informatics* 14.11 (2018), pp. 4724–4734 (cit. on p. 17).

[Ste10] W. Steiner. “An evaluation of SMT-based schedule synthesis for time-triggered multi-hop networks”. In: *2010 31st IEEE Real-Time Systems Symposium*. IEEE. 2010, pp. 375–384 (cit. on p. 23).

[TG08] M. D. J. Teener, G. M. Garner. “Overview and timing performance of IEEE 802.1 AS”. In: *2008 IEEE international symposium on precision clock synchronization for measurement, control and communication*. IEEE. 2008, pp. 49–53 (cit. on p. 20).

[TV99] E. Tovar, F. Vasques. “Real-time fieldbus communications using Profibus networks”. In: *IEEE transactions on Industrial Electronics* 46.6 (1999), pp. 1241–1251 (cit. on p. 19).

[Var01] A. Varga. “Discrete event simulation system”. In: *Proc. of the European Simulation Multiconference (ESM’2001)*. 2001, pp. 1–7 (cit. on pp. 18, 21).

[VBHT22] M. Vlk, K. Brejchová, Z. Hanzálek, S. Tang. “Large-scale periodic scheduling in time-sensitive networks”. In: *Computers & Operations Research* 137 (2022), p. 105512 (cit. on p. 23).

[VH10] A. Varga, R. Hornig. “AN OVERVIEW OF THE OMNeT++ SIMULATION ENVIRONMENT”. In: ICST, May 2010. DOI: 10.4108/ICST.SIMUTOOLS2008.3027 (cit. on pp. 18, 21).

[VHT21] M. Vlk, Z. Hanzálek, S. Tang. “Constraint programming approaches to joint routing and scheduling in time-sensitive networks”. In: *Computers & Industrial Engineering* 157 (2021), p. 107317 (cit. on p. 23).

[Wax88] B. M. Waxman. “Routing of multipoint connections”. In: *IEEE journal on selected areas in communications* 6.9 (1988), pp. 1617–1622 (cit. on p. 35).

[WG02] L. A. Wallis, I. Greaves. “Injuries associated with airbag deployment”. In: *Emergency medicine journal* 19.6 (2002), pp. 490–493 (cit. on p. 17).

All links were last followed on July 23, 2024.

Declaration

I hereby declare that the work presented in this thesis is entirely my own and that I did not use any other sources and references than the listed ones. I have marked all direct or indirect statements from other sources contained therein as quotations. Neither this work nor significant parts of it were part of another examination procedure. I have not published this work in whole or in part before. The electronic copy is consistent with all submitted copies.

place, date, signature