Downloaded from orbit.dtu.dk on: May 07, 2024

Safety- and Security-Aware Configuration Synthesis for Time-Sensitive Networking

Reusch, Niklas

Publication date:
2022

Document Version
Publisher's PDF, also known as Version of record

Link back to DTU Orbit

Citation (APA):
Reusch, N. (2022). *Safety- and Security-Aware Configuration Synthesis for Time-Sensitive Networking*. Tecnical University of Denmark.

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

- Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
- You may not further distribute the material or use it for any profit-making activity or commercial gain
- You may freely distribute the URL identifying the publication in the public portal

If you believe that this document breaches copyright please contact us providing details, and we will remove access to the work immediately and investigate your claim.

Safety- and Security-Aware Configuration Synthesis for Time-Sensitive Networking

Niklas Reusch

Kongens Lyngby 2022

Technical University of Denmark
Department of Applied Mathematics and Computer Science
Richard Petersens Plads, building 324,
2800 Kongens Lyngby, Denmark
Phone +45 4525 3031
compute@compute.dtu.dk
www.compute.dtu.dk

Summary (English)

In the past decades, more and more areas of human life have become influenced by networked cyber-physical systems (CPS). Increasingly, we trust these systems to execute critical functions, such as controlling our cars and airplanes and managing dangerous processes in factories and energy systems. Hence, these CPS have stringent safety, real-time, and security requirements. In this thesis, we consider CPS that are using Time-Sensitive Networking (TSN) for communication. The IEEE 802.1 TSN standardization is developing a "toolbox" of many standards that extends Ethernet for safety-critical and real-time applications in several areas, e.g., automotive, aerospace, or industrial automation. TSN-based distributed CPS are composed of end-systems interconnected by network switches and duplex physical links; in TSN, communication streams from safety-critical and real-time applications can share the same communication channel with less-critical streams safely. However, the flexibility of TSN comes at the high price of a huge and poorly understood configuration space. TSN has many "configuration knobs", that decide, e.g., the real-time transmission of critical traffic via so-called Gate Control List (GCL) schedules, the stream priorities and their assignment to queues, and the routing of streams on disjoint paths to achieve fault-tolerance.

Most TSN scheduling mechanisms are designed for homogeneous TSN networks, in which all network devices must have at least the TSN capabilities related to scheduled gates and time synchronization. However, this assumption is often unrealistic since many distributed applications use heterogeneous TSN networks with legacy or off-the-shelf end-systems that are unscheduled and/or unsynchronized. In this thesis, we first propose a new scheduling paradigm for heterogeneous TSN networks that intertwines a network calculus worst-case interference analysis within the scheduling step. Thus, we support heterogeneous TSN networks featuring unscheduled and/or unsynchronized end-systems while guaranteeing the real-time properties of critical communication. Se-

curity is an important requirement in distributed CPS. We highlight the importance of addressing security at the same time with safety and timing requirements. We consider the Timed Efficient Stream Loss-Tolerant Authentication (TESLA) low-resource multicast authentication protocol to guarantee the security requirements, and redundant disjunct message routes to tolerate link failures. Given a TSN-based distributed CPS, a set of applications with tasks and messages, as well as a set of security and redundancy requirements, in the second part of the thesis we are interested to synthesize a system configuration such that the real-time, safety, and security requirements are upheld.

TSN is used within the computing continuum, from interconnecting IoT devices to the networks used in Edge Computing and Cloud Computing data centers. However, as systems become larger and more interconnected, the threat level increases and untrusted devices pose high security risks. Hence, in the final part of the thesis, we consider the use of Remote Attestation (RA) to authenticate the functionality of a remote device, thus, allowing for the provision of strong assurance guarantees. We propose solutions for the automatic management of resources in the IoT to Edge Computing continuum to integrate dynamic Edge applications with safety and security-critical real-time applications. We show that our approach generates dependable configurations that can meet the timing constraints of critical applications, have enough resources to perform RA for security, and can accommodate Edge applications.

The configuration synthesis challenges tackled in the thesis form intractable combinatorial optimization problems. We have used a variety of optimization algorithms, from problem-specific heuristics to metaheuristics such as Simulated Annealing, and exact methods such as Constraint Programming, to tackle these problems. These approaches are evaluated on synthetic and realistic test cases of different sizes, and their advantages and disadvantages are discussed and compared to the related work. The approaches proposed in the thesis have been implemented as open-source software prototypes and have been validated via simulations.

Summary (Danish)

I de seneste årtier er flere og flere områder af menneskers liv blevet mere afhangige af og påvirkede fra cyber-physical systems (CPS). Vi stoler i stigende grad på, at disse systemer udfører kritiske funktioner, såsom kontrol af vores biler og fly og styring af farlige processer i fabrikker og energisystemer. Derfor har disse CPS strenge safety-, realtids- og securitykrav. Fokus for denne afhandling er CPS, der bruger Time-Sensitive Networking (TSN). IEEE 802.1 TSN- standardiseringen udvikler en "værktøjskasse"med mange standarder, der udvider Ethernet til sikkerhedskritiske og realtidsapplikationer på flere områder, f.eks. bilindustrien, rumfart eller industriel automation. TSN-baserede distribuerede CPS er sammensat af end-systems forbundet med netværksswitches og duplex fysiske links; i TSN kan kommunikationsstrømme fra sikkerhedskritiske og realtidsapplikationer dele kommunikationskanal med mindre kritiske strømme. Fleksibiliteten ved TSN kommer dog til den høje pris af en stor og dårligt forstået configuration space. TSN har mange "konfigurationsknapper", som f.eks. bestemmer realtidstransmissionen af kritisk trafik via såkaldte Gate Control List (GCL), streamprioriteter og deres tildeling til køer og routing af streams på usammenhængende stier for at opnå fejltolerance.

De fleste TSN scheduling mekanismer er designet til homogene TSN-netværk, hvor alle netværksenheder skal have mindst TSN-kapaciteter relateret til scheduled gates og tidssynkronisering. Denne antagelse er dog ofte urealistisk, da mange distribuerede applikationer bruger heterogene TSN-netværk med ældre eller off-the-shelf end-systems, der er unscheduled og/eller usynkroniserede. I denne afhandling foreslår vi først et nyt scheduling methode for heterogene TSN-netværk, der bruger en network-calculus worst-case interferensanalyse inden for planlægningstrinnet. Så støtter vi heterogene TSN-netværk med unscheduled og/eller usynkroniserede end-systems, mens vi garanterer realtidsegenskaberne for kritisk kommunikation. Security er et vigtigt krav i di-

stribueret CPS. Vi viser, at det er meget gavnligt at adressere security samtidig med krav til safety og timing. Vi overvejer TESLA (Timed Efficient Stream Loss-Tolerant Authentication) lavressource multicast-autheticationprotokol for at garantere security og redundante disjunkte beskedruter til at tolerere linkfejl. Givet en TSN-baseret distribueret CPS, et sæt applikationer med tasks og beskeder, samt et sæt security- og redundanskrav, er vi i anden del af afhandlingen interesseret i at syntetisere en systemkonfiguration, således at real-time, safety, og securitykrav overholdes.

TSN bruges inden for computing continuum, fra sammenkobling af IoT-enheder til netværk brugt i Edge Computing og Cloud Computing-datacentre. Men efterhånden som systemerne bliver større og mere sammenkoblede, stiger trusselsniveauet, og upålidelige enheder udgør højere sikkerhedsrisici. Derfor overvejer vi i den sidste del af afhandlingen brugen af Remote Attestation (RA) til at autentificere funktionaliteten af en enhed, hvilket giver mulighed for at give stærke assurance garantier. Vi foreslår løsninger til automatisk styring af ressourcer i IoT til Edge Computing-continuum for at integrere dynamiske Edge-applikationer med safety- og securitykritiske realtidsapplikationer. Vi viser, at vores tilgang genererer dependable konfigurationer, der kan opfylde tidsbegrænsningerne for kritiske applikationer, har nok resourcer til at udføre RA for sikkerheden og kan rumme Edge-applikationer.

Konfigurationsudfordringerne, der tages op i afhandlingen, danner intractable kombinatoriske optimeringsproblemer. Vi har brugt en række forskellige optimeringsalgoritmer, lige fra problemspecifikke heuristika til metaheuristik såsom Simulated Annealing, og eksakte metoder såsom Constraint Programming, til at tackle disse problemer. Disse algoritmer evalueres på syntetiske og realistiske testcases af forskellig størrelse, og deres fordele og ulemper diskuteres og sammenlignes med relaterede arbejde. De foreslåede tilgange i afhandlingen er implementeret som open source software prototyper og er blevet valideret via simuleringer.

Preface

This thesis was prepared at Department of Applied Mathematics and Computer Science (DTU Compute) to fulfill the requirements for acquiring a Ph.D. degree in Computer Science.

In this thesis, I propose methods to aid the configuration of safety-critical real-time systems using Time-Sensitive Networking. I investigated how security protocols can be considered at the design stage and showed that this has advantages over considering them separately. The thesis consists of an introductory chapter and three papers.

The work has been supervised by Professor Paul Pop and co-supervised by Professor Nicola Dragoni.

Lyngby, 31-December-2022
Niklas Reusch

Acknowledgments

First of all, I would like to thank Paul for his amazing work as a main supervisor throughout the years. All the way from when I took his course on fault-tolerant systems to the end of this PhD, he has always been incredibly responsive, helpful and supportive. Many times I went into a meeting frustrated with a problem or lost for direction, and came out with a solution and a smile on my face. His support always kept me going. This thanks also extends to the whole research group: Thank you, Reza, Bahram, Luxi and Voica for great collaboration and introducing me when I was new!

Secondly, I would like to thank my co-supervisor Nicola for introducing me into the world of security research. A special thanks goes to my frequent co-author Silviu for being an unofficial co-supervisor, great collaborator and host in Vienna! Thanks to my peers Koen, Michele, Emil, Freja, Manja and Hjørdis for the good times at DTU and helping me in various ways. Thanks to Kim for his hard work in organizing events for PhD students and fostering the social life in the entire department.

Lastly, I would like to thank my family and friends. Thanks to my parents and sister for always being there for me and encouraging me to pursue my interests. Thanks to Dr. Peter Richter for believing in my talent and teaching me many life-lessons, and to Jutta Roßbach for her incredible support during my high-school university studies. And a last special thanks to Jonas and Jill for their moral support and hosting me in difficult corona times.

I am very grateful to all of you!
Niklas

Abbreviations

ASAP	As-Soon-As-Possible
ALAP	As-Late-As-Possible
CP	Constraint Programming
CPS	Cyber-Physical System
DAG	Direct Acyclic Graph
ED	Edge Device
ECP	Edge Computing Platform
GA	Genetic Algorithms
GRASP	Greedy Randomized Adaptive Search Procedures
GCL	Gate-Control List
ILP	Integer Linear Programming
MAC	Message Authentication Code
OMT	Optimization Modulo Theories
RA	Remote Attestation
SA	Simulated Annealing
SIL	Safety Integrity Level
SMT	Satisfiability Modulo Theories
TAS	Time-Aware Shaper
TESLA	Timed Efficient Stream Loss-Tolerant Authentication
TSN	Time-Sensitive Networking
WCET	Worst-case Execution Time

List of Figures

1.1 TSN Switch model and Gate Control List. 5
1.2 Time-Aware Shaper Determinism Problem (Inspired by [CSCS16]). . 6
1.3 Comparison of Symmetric and Asymmetric Authentication. 9
1.4 Remote Attestation Process (Inspired by [SL16]). 10

2.1 TSN Switch model and Queue Interference. 28
2.2 Example Window Configurations. 31
2.3 Overview of our CPWO Optimization Strategy. 33
2.4 Example Capacity for a Window. 36
2.5 Example Capacity and Transmission Demand for a Window. 37
2.6 Network Topologies used in the Test Cases [ZPS17]- 44
2.7 WCD vs. Simulated Delays. 48
2.8 Objective Value Boxplots for Medium and Large SAWO Test Batches. 52
2.9 Objective Value Boxplots for Huge SAWO Test Batches. 53

3.1 Automotive TSN-based CPS with Redundant Routes. 57
3.2 Simplified TSN Switch Representation. 59
3.3 TESLA Key Chain (adapted from [PCTS02]). 62
3.4 Example Architecture and Application Models. 65
3.5 Example Security Model for the Applications in Figure 3.4. 67
3.6 Example Solution Schedules for the Models in Figure 3.4. 69
3.7 Example Precedence Graph with Associated Order. 79
3.8 Backtrack Example: Scheduling s_2. 85
3.9 Feasible Region Example. 87
3.10 Example Latency Optimization for Secure Streams. 88
3.11 Scalability Results of CP Solution. 93

4.1 Architecture of the Edge Computing Platform. 104

4.2 TSN Switch. 105
4.3 Critical Application Example. 107
4.4 Edge Application Example. 108
4.5 RA Application Example. 109
4.6 Example Solution Schedules. 110

A.1 TSNConf: Network Architecture Visualization. 120
A.2 TSNConf: Various Visualizations. 121
A.3 TSNConf: Routing visualization. 121
A.4 TSNConf: Schedule Visualization 122

List of Tables

1.1 Relevant TSN Standards and Amendments. 4

2.1 Scheduling Approaches in TSN. 23
2.2 Summary of System Model Notations. 27
2.3 Definition of Terms for CP Model Formulation. 34
2.4 Details of the Synthetic Test Cases. 45
2.5 CPWO Evaluation Results on Synthetic Test Cases. 46
2.6 CPWO Results on Realistic Test Cases. 47
2.7 Scalability Evaluation of CPWO. 48
2.8 SAWO Evaluation Results on Synthetic Test Cases. 50
2.9 SAWO Results on Realistic Test Cases. 50
2.10 Parameters of SAWO Test Batches. 51

3.1 System Model Notations. 64
3.2 Matrix X for Example from Section 3.5.1. 70
3.3 Results of Scalability Tests. 92
3.4 Impact of Security and Redundancy Measures. 94

4.1 Evaluation Results. 116
4.2 Impact of Extensibility Formulation on Average Latency of Edge Applications. 116

Contents

Summary (English) **i**

Summary (Danish) **iii**

Preface **v**

Acknowledgments **vii**

Abbreviations **ix**

1 Introduction **1**
- 1.1 Motivation 1
- 1.2 Time-Sensitive Networking 3
- 1.3 Safety and Security Threats and Protections 6
  - 1.3.1 Safety Protections 7
  - 1.3.2 Security Protections 7
- 1.4 Related Work 11
- 1.5 Thesis Overview and Contributions 14

2 Paper A: Configuration Optimization for Heterogeneous Time-Sensitive Networks **17**
- 2.1 Introduction 18
- 2.2 Related Work 20
  - 2.2.1 In-depth Formal Comparison 22
- 2.3 System Model 27
  - 2.3.1 Network Model 27
  - 2.3.2 Switch Model 28
  - 2.3.3 Application Model 29
- 2.4 Problem Formulation 30

2.4.1 Motivational Example 30
2.5 Constraint Programming Window Optimization 32
2.5.1 CP Objective Function 34
2.5.2 Variables 34
2.5.3 Constraints 35
2.5.4 Timing Constraints 36
2.6 Simulated Annealing Window Optimization 40
2.6.1 Initial Solution 41
2.6.2 SA Objective Function 42
2.6.3 Neighbor Function 43
2.7 Evaluation 44
2.7.1 CPWO Evaluation 44
2.7.2 SAWO Evaluation 49
2.8 Conclusions 53

3 Paper B: Dependability-Aware Routing and Scheduling for Time-Sensitive Networking 55
3.1 Introduction 56
3.1.1 Related Work 57
3.1.2 Contributions 58
3.2 Time-Sensitive Networking 59
3.3 Timed Efficient Stream Loss-Tolerant Authentication 61
3.4 System Models 63
3.4.1 Architecture Model 63
3.4.2 Application Model 63
3.4.3 Fault Model 66
3.4.4 Threat Model 66
3.4.5 Security Model 66
3.5 Problem Formulation 68
3.5.1 Motivational Example 68
3.6 Constraint Programming Formulation 70
3.6.1 Optimizing Redundant Routing 70
3.6.2 Optimizing P_{int} 73
3.6.3 Optimizing Scheduling 74
3.7 Metaheuristic Formulation 78
3.7.1 Precedence Graph 78
3.7.2 Initial Solution 80
3.7.3 Neighbourhood Function 80
3.7.4 Cost Function 82
3.7.5 ASAP List Scheduling 82
3.7.6 Optimizing the Latency for Secure Streams 87
3.8 Experimental Results 90
3.8.1 Test Cases 90
3.8.2 Scalability Evaluation 91

3.8.3 Impact of Adding Redundancy and Security to a Test Case . . 93
3.8.4 Discussion 95
3.9 Conclusion 95
3.10 Appendix A: Routing Constraint Formulation for Allowed Overlap . . 96
3.11 Appendix B: More Functions from Metaheuristic Formulation 96
3.11.1 CalculateLowerBound 96
3.11.2 UpdateSchedule 98

4 Paper C: Mapping and Scheduling Real-Time Applications on Edge Computing Platforms with Remote Attestation for Security 99
4.1 Introduction 100
4.1.1 Related Work 102
4.1.2 Contributions 103
4.2 Edge Computing Platform Model 103
4.2.1 Time-Sensitive Networking (TSN) 105
4.2.2 Remote Attestation 106
4.3 Application Models 106
4.3.1 Critical Application Model 107
4.3.2 Edge Application Model 108
4.3.3 Remote Attestation Model 108
4.4 Problem Formulation 109
4.4.1 Example 109
4.5 Optimization Strategy 111
4.5.1 Routing Constraints 111
4.5.2 Task Constraints 113
4.5.3 Stream Constraints 114
4.6 Experimental Evaluation 115
4.7 Conclusions and Future Work 117

A TSNConf: Testcase and Schedule Visualization Tool 119

Bibliography 123

CHAPTER 1

Introduction

1.1 Motivation

Safety-critical systems are those whose failure may have catastrophic consequences, such as environmental or economic damage or even loss of life [Kni02]. Cyber-Physical System (CPS) are those that combine computational capabilities with the ability to interact with the physical world [BG11], and thus can be potentially dangerous to humans. An example could be a computer-controlled robotic arm in a factory or a lane-keeping assistant in a vehicle. Hence, many CPS are also safety critical. With the replacement of mechanical controllers by hardware and software, and the increase in computing power, more and more advanced applications become possible. Nowadays, CPS are distributed, with many components, enabling autonomous driving, assisted surgeries and efficient energy grids [SWYS11, BG11, Wol09]. Unfortunately, the improvements to quality of life or other areas, that these systems offer, are not without risk. There have been cases in which improper systems design and configuration have led to serious consequences, for example, during the Ariane V launch failure [ESA96] or the crash of the London city ambulance dispatch system [Kni02]. In consequence, properly configuring and protecting these systems is of the highest priority.

A careful planning process is necessary for the engineering and operation of a safety-critical system. The IEC 61508 functional safety standard [Int10], which has been an inspiration to many other industry-specific standards, defines a life cycle of a sys-

tem split into fifteen individual phases [Rau14, SS11], of which we summarize the most important ones for this work. The cycle begins with risk assessment stages, in which potential hazards are identified, the risk of these appearing quantified and safety-requirements to mitigate these risks are determined. Functions which are necessary to safeguard the system are determined and assigned a Safety Integrity Level (SIL), based on their expected reliability. In the next step, the system's hardware and software is designed based on the identified requirements and validated by intensive testing.

In many safety-critical systems, timing is of great importance, especially in networked systems, where many components have to collaborate. For example, an important message like the brake signal in a vehicle, or the emergency stop signal at a factory machine, should have a bounded latency, that is, arrive before a certain deadline and not be unexpectedly delayed or lost. In some applications, the variation in arrival time between different instances of the same message, called jitter, is also critical [MAS+18]. The correctness of these systems' behavior depends on the timing: A brake signal arriving ten seconds late is worthless and may be catastrophic. Such systems are also called real-time systems [KS22]. In real-time systems, there is a distinction between soft and hard deadlines, where missing a soft deadline is sometimes acceptable, but decreases the systems' usefulness, while missing a hard deadline may have catastrophic consequences [KS22].

Many real-time systems use special operating systems and networking technologies, which differ from those used in the consumer market. While consumer technologies are optimized for speed and efficiency, they lack the mechanisms for providing the guarantees necessary in safety-critical and real-time systems [Dec05]. For example, to be able to guarantee timing requirements, real-time technologies provide hardware that allows a designer to specify schedules, i.e., timetables which specify when which message is sent and when which task is executed. The construction of such schedules, as well as other associated optimization problems such as task mapping or routing, are typically NP-hard, since the associated decision-problem is NP-complete. [Ste10, SOLM22] One such real-time technology, which is the focus of this thesis, is Time-Sensitive Networking (TSN), which allows the scheduling of messages in a network by synchronizing devices and employing networking switches which can selectively forward messages of different priorities using Gate-Control Lists (GCLs). More details on TSN can be found in Section 1.2.

Historically, many safety-critical systems were disconnected from the outside world, which gave them an extra layer of protection from malicious attacks from outsiders, and which lead to research focusing on safety and reliability for those systems. However, with the advent of, for example, smart cars, smart grids or Industry 4.0, there are safety-critical systems that are inherently federated and rely on connection to other systems or the internet for their functionality. Unfortunately, this also makes it easier for external entities to attack these systems, with potentially catastrophic consequences [KK12].

An example of an attack on critical infrastructure is the Maroochy water services breach, in which a disgruntled ex-contractor managed to hack into the control systems of the company and cause one million liters of untreated sewage to enter local waterways [SM07]. Another famous example of an attack on critical infrastructure is the Stuxnet malware discovered in 2010 [CAN11]. It was probably targeted at Iran's nuclear program and managed to infect an air-gapped (disconnected from the internet) nuclear enrichment facility via the use of USB-flash drives. It exploited multiple zero days (undiscovered vulnerabilities) to be able to stealthily change the frequency of connected industrial drives, e.g., nuclear centrifuges, and thus damage or destroy them [CAN11]. A more recent example are the sophisticated attacks on Ukraine's energy grid in 2015 and 2016 that left more than 225.000 customers without power [Lee17].

Also in Denmark, there have been successful attacks on critical-infrastructure: Recently, an attack on an external supplier of critical software lead to a complete standstill of the national train network [Tro22]. Other attacks managed to disrupt the operations of the public service company Kalundborg Forsyning [Hau22] and the energy provider Vestas [NMFMMT22]. A recent report by the national audit agency found that many society-critical systems are not prepared for computer system breakdowns or cyberattacks [AJP$^+$22].

Consequently, security has become an important aspect in designing safety-critical networks. Integrating security in the design process is not straightforward, since the security measures come with their own requirements, which have to be carefully assessed and integrated, considering the other strict timing and reliability requirements of the network. We show the importance of considering security in conjunction with other safety and reliability measures at the design-stage of a safety-critical system. We consider the Timed Efficient Stream Loss-Tolerant Authentication (TESLA) protocol as an example solution for message authentication, see Chapter 3, and the use of Remote Attestation (RA) as an example solution to check the integrity of a remote device, see Chapter 4.

1.2 Time-Sensitive Networking

Time-Sensitive Networking (TSN) is a set of standards and amendments to standards developed by an IEEE 802.1 working group [Ins16c]. The goal of these standards is to provide deterministic connectivity, i.e., guaranteed packet transport with bounded latency and low packet delay variation (jitter) for previously non-deterministic IEEE 802 networks, e.g., those using 802.3 Ethernet. There are several proprietary technologies, that offer similar functionality, e.g., TTEthernet (SAE AS6802 [Iss11, SBHP11]), PROFINET [Comb], and EtherCAT [Coma]. However, TSN is an open and flexible

standard, provides high bandwidth for modern applications and allows multiple traffic classes on the same wire [DXL+23, Ins16c, SOLM22].

To achieve these goals, TSN defines several "sub-standards" to IEEE 802.1Q [Ins14], introducing new mechanisms for Ethernet bridges, extensions to the IEEE 802.3 Media Access Control (MAC) layer, and introducing new protocols. It is not necessary to use all sub-standards at the same time, rather they should be chosen based on the systems requirements [SHM+21, LBS19]. TSN gives a system designer a large flexibility at the cost of a complex configuration problem, where the designer has to decide which sub-standards to use and how to configure each of them [LBS19]. Table 1.1 lists the most relevant standards and amendments for our work.

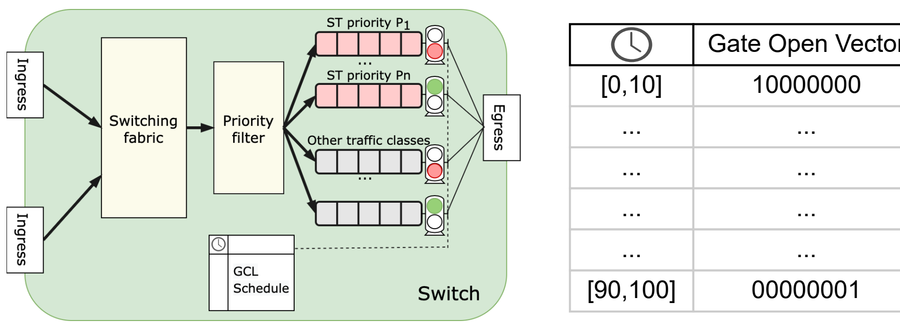
Standard	**Description**
802.1AS-Rev [Ins17a]	Timing and Synchronization for Time-Sensitive Applications
802.1CB [Ins17b]	Frame Replication and Elimination for Reliability
802.1Qbv [Ins16b]	Enhancements for Scheduled Traffic, Time Aware Shaper
802.1Qbu [Ins16a]	Frame Preemption
802.1Qci [Ins17c]	Per-Stream Filtering and Policing

Table 1.1: Relevant TSN Standards and Amendments.

The fundamental mechanisms that enable deterministic temporal behavior over Ethernet are, on the one hand, the clock synchronization protocol defined in IEEE 802.1AS-rev [Ins17a], which provides a common clock reference with bounded deviation for all nodes in the network, and on the other hand, the timed-gate functionality (IEEE 802.1Qbv [Ins16b]) enhancing the transmission selection on egress ports. The timed-gate functionality enables the predictable transmission of communication streams according to the predefined times encoded in schedules called Gate-Control Lists (GCL). A stream in TSN is a communication carrying a certain payload size from a talker end-system (sender) to one or multiple listener end-systems (receivers), which are connected by switches. A stream may or may not have timing requirements. Critical streams may have maximum end-to-end latency and jitter requirements and are often transmitted periodically.

Other amendments within TSN (c.f. [Ins16c]) provide additional mechanisms that can be used either in conjunction with 802.1Qbv or standalone. IEEE 802.1CB [Ins17b] enables stream identification, based on, e.g., the destination MAC and VLAN-tag fields in the frame, as well as frame replication and elimination for redundant transmission. IEEE 802.1Qbu [Ins16a] enables preemption modes for mixed-criticality traffic, allowing express frames to preempt lower-priority traffic. IEEE 802.1Qci [Ins17c] defines frame metering, filtering, and time-based policing mechanisms on a per-stream basis using the stream identification function defined in 802.1CB.

We detail the Time-Aware Shaper (TAS) mechanism defined in IEEE 802.1Qbv [Ins16b]



(a) TSN Switch Internals.

🕓	Gate Open Vector
[0,10]	10000000
...	...
...	...
...	...
...	...
[90,100]	00000001

(b) Gate-Control List

Figure 1.1: TSN Switch model and Gate Control List.

via the simplified representation of a TSN switch in Figure 1.1a. The figure presents a scenario in which communication received on one of two available ingress ports will be routed to an egress port. The switching fabric will determine, based on internal routing tables and stream properties, to which egress port a frame belonging to the respective stream will be routed (in our logical representation, there is only one egress port). Each port will have a priority filter that determines which of the available 8 traffic classes (priorities) of that port the frame will be enqueued in. This selection will be made based on either the PCP field of the 802.1Q VLAN-tag of frames or the *stream gate instance table* of 802.1Qci, which can be used to circumvent traffic class assignment of the PCP code.

As opposed to regular 802.1Q bridges, where the transmission selection sends enqueued frames according to their respective priority, in 802.1Qbv bridges, there is a TAS mechanism, providing timed-gates, associated with each traffic class queue and positioned before the transmission selection algorithm (here depicted as a traffic light). A timed-gate can be either in an *open (o)* or *closed (C)* state. When the gate is open, traffic from the respected queue is allowed to be transmitted, while a closed gate will not allow the respective queue to be selected for transmission, even if the queue is not empty. The state of the queues is encoded in a local schedule (the GCL), depicted in Figure 1.1b. Each entry defines a time interval and a bitvector, indicating which gates are open and closed. Hence, whenever the local clock reaches the beginning of the next time interval, the timed-gates will be changed to the respective open or closed state. If multiple non-empty queues are open simultaneously, the transmission selection selects the queue with the highest priority for transmission. The GCL has a limited number of entries, so care should be taken to create schedules that do not require too many entries [SOLM22].

The Time-Aware Shaper functionality of 802.1Qbv, together with the synchronization protocol defined in 802.1ASrev, enables a global communication schedule that orches-

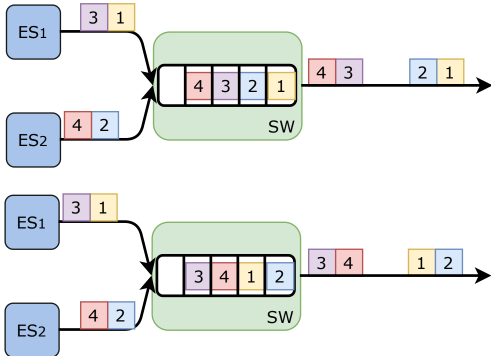


Figure 1.2: Time-Aware Shaper Determinism Problem (Inspired by [CSCS16]).

trates the transmission of frames across the network such that real-time constraints (usually end-to-end latencies) are fulfilled. Craciunas et al. [CSCS16] define correctness conditions for generating GCL schedules, resulting in a strictly deterministic transmission of frames with 0 jitter. Apart from the technological constraints, e.g., only one frame transmitted on a link at a time, the deterministic behavior over TSN is enforced in [CSCS16] through isolation constraints. Since the TAS determines the temporal behavior of entire traffic classes and not of individual frames, the queue state may become non-deterministic if multiple streams share the same queue and arrive at similar times, see Figure 1.2, where minor differences in arrival time could affect the queuing order. Thus, [CSCS16] proposes strict isolation of critical streams by not allowing two critical streams to be enqueued in the same queue or at the same time. This restriction can be relaxed if the possible overlap is accounted for in the latency calculations, as shown in [SOCS18] or Chapter 2.

TSN offers the tools to provide many of the guarantees necessary for safety-critical real-time networks. However, it also comes with a huge amount of different sub-standards and their configuration, in which the system designer has to make the right choices. This is only possible if the designer is aware of possible threats and methods to counter them. Some of these threats and counters are described in the next section.

1.3 Safety and Security Threats and Protections

In this chapter, we describe some possible threats to safety-critical networks and how to protect against them. Hereby, we differentiate between non-deliberate threats in Section 1.3.1 and deliberate threats in Section 1.3.2.

1.3.1 Safety Protections

One important threat to safety-critical networks are *faults*, a network addressing these faults is called *fault-tolerant*. The authors in [NEL90] give the terms *fault*, *error* and *failure* a specific meaning: A *fault* is an anomalous condition of the system, caused by physical damage, radiation, electromagnetic interference, but also by implementation mistakes. An *error* is the manifestation of a fault in the system, leading to an unintended logical state. A fault does not necessarily have to result in an error. If an error leaves the systems unable to perform its function, we speak of a *failure*. Faults can be categorized by their duration: A *transient fault* occurs once and disappears. An *intermittent fault* occurs in regular intervals. A *permanent fault* never disappears.

A system designer should anticipate faults and configure the system, such that they cannot turn into an error or failure. *Dependability* is the measure of how well a system can perform its intended function in the presence of faults [NEL90]. It can be quantified deterministically (the system tolerates X faults) or probabilistically (the system can perform its function at time Y with a probability Z). This probabilistic measure is also called *reliability*.

To improve the reliability of a system, a lot of different techniques are used. A fundamental technique is redundancy in hardware and software [NEL90]. The same processing unit, cable, or function can be replicated multiple times. When a fault occurs in one unit, it can be *masked* by the correct functionality of the other units. To decide which result is the correct one, if the faulty unit forwards a wrong result, there can be a *voting* mechanism in which only the result with the most votes is forwarded. Sometimes it is also possible for a unit to self-detect its faulty state and prevent its result from being forwarded. Another technique is the detection of errors, followed by diagnosis, containment, and repair. Some error can be corrected immediately, e.g., using error-correcting codes. Others may be corrected by replacing the faulty part or by reconfiguring the system to avoid it.

1.3.2 Security Protections

Another large threat to safety-critical networks are deliberate attacks, which *security* measures attempt to prevent. Already during the initial risk assessment stage of safety-critical product development should security be considered. A common tool used in security research are *threat models* [Sho14]. The idea is to consider the product being built and theorize what an attacker could do given certain capabilities, to then decide on countermeasures.

Fundamental Goals

It is difficult to clearly divide security goals and solutions into separate categories. However, three fundamental goals in security that are often mentioned are *confidentiality*, *integrity*, and *availability* [CWW+16].

The goal of *confidentiality* is to ensure that only authorized entities can see a piece of information. Methods to achieve confidentiality include encryption, authorization, and access control, including physical access [GT14]. Some examples of confidentiality measures in safety-critical networks are presented in [MYPB14] and [JEP12], where the authors propose a scheduling scheme to prevent unauthorized tasks from reading information from other tasks and use message-encryption to prevent an eavesdropper on the network from learning the content of messages.

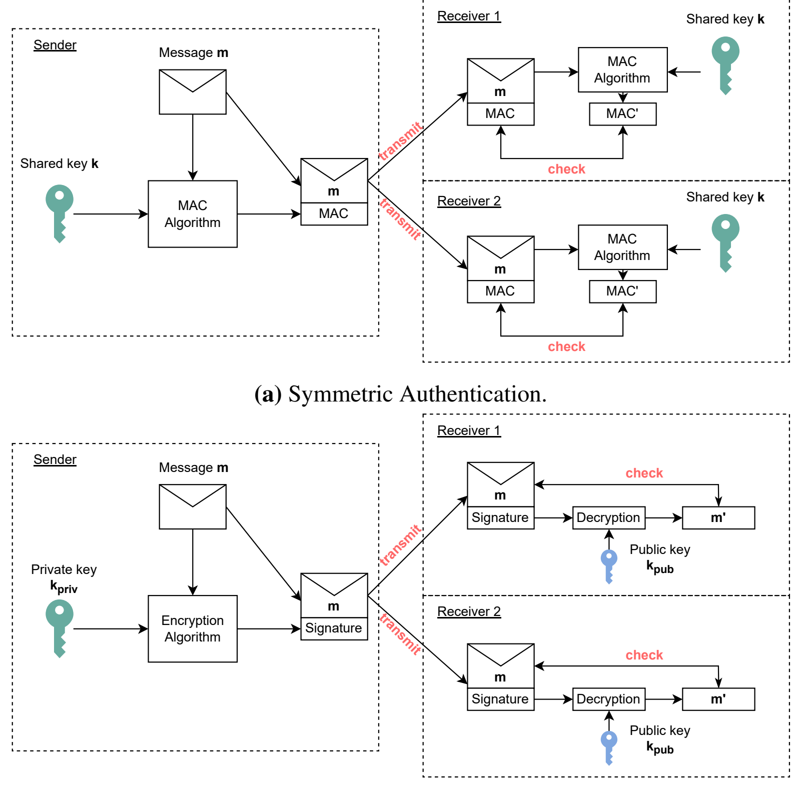
Integrity is the concept that only authorized entities should be able to create or modify information [CWW+16]. Many methods to provide integrity are also used for fault-tolerance purposes, as they protect both again faults and deliberate attacks. Some examples include checksums, error-correcting codes, backups, and redundancy [GT14]. A method to provide integrity that is unique to security is *authentication*. It attempts to ensure that an entity is who they claim to be, and that its communication is genuine [CWW+16, GT14]. *Remote Attestation* is another security method, which enables the remote checking of a device's integrity. We present these methods in more details in Section 1.3.2 and Section 1.3.2, since we use them to provide security guarantees in Chapter 3 and Chapter 4.

The last fundamental goal of security is *availability*, the notion that systems and information are accessible and functional [GT14]. Methods to ensure availability include physical protection, redundancy, backups, and traffic filtering. Attacks against availability include, for example, physical destruction or Denial of Service (DoS) attacks, where a system is flooded with fake messages. Much of the work that considers redundancy is also applicable here, possibly in conjunction with remote attestation to identify misbehaving end-systems. There is also, for example, some work that uses 802.1Qci filtering abilities to protect against DoS in TSN [MHKS19].

Authentication

In the context of secure network communication, it is a common assumption that a so called *man-in-the-middle* can listen to and modify messages between two end-systems. In this case, we want to ensure that a modified message is not accepted at the listener, which is called message authentication [MVOV96, Chapter 9].

The most commonly used schemes for message authentication are *message authentication codes (MACs)* and *digital signatures* [MVOV96]. A MAC is calculated on a given message using a shared secret key and a cryptographic primitive, e.g., a cryptographic hash function, and appended to the message. Without the knowledge of the key, it should be infeasible to compute the correct MAC for a given message. An end-system, which possesses the secret key, can thus verify that a received message is genuine, by calculating the MAC of the message and comparing it to the received MAC, as can be seen in Figure 1.3a. This is called a *symmetric* scheme, since the same key is used at both ends. Digital signatures are an example of an *asymmetric* scheme, where different keys are used at the sender and the receiver. The sender possesses a secret private key, with which he calculates a signature, which is appended to the message. The receivers possess a public key, which allows them to decrypt the signature, without being able to create the same signature themselves, as can be seen in Figure 1.3b.



(a) Symmetric Authentication.

(b) Asymmetric Authentication.

Figure 1.3: Comparison of Symmetric and Asymmetric Authentication.

The advantage of symmetric authentication is its simplicity and efficiency, but is prob-

lematic in a setting with many end-systems [BDF01]. If all end-systems share the same key, just one compromised end-system would allow an attacker to impersonate all other end-systems. To avoid this, a different key for each end-system pair could be used. However, this comes with a storage overhead, and is infeasible in a multicast setting, where the same message is sent to a lot of receivers. In that case, the size of the message would grow linearly with the number of receivers, as a different MAC would have to be appended for each receiver. Asymmetric authentication avoids these problems, but comes at the cost of a high bandwidth and computation overhead [PCTS02, MVOV96]. A hybrid scheme that works well for multicast authentication is Timed Efficient Stream Loss-Tolerant Authentication (TESLA), which we explain in detail in Chapter 3. It uses efficient symmetric authentication with MACs, but uses time for asymmetry, by keeping the key secret until the associated message has arrived. It relies on time synchronization between end-systems, which is available in TSN.

Remote Attestation

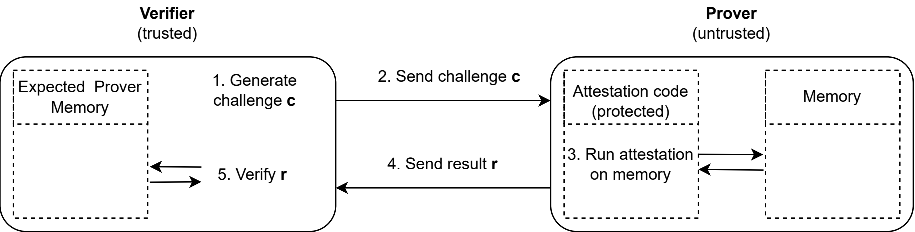


Figure 1.4: Remote Attestation Process (Inspired by [SL16]).

Remote Attestation (RA) is a mechanism to ensure the integrity of devices in a network [SL16]. Using RA, a trusted device called *verifier* can check if an untrusted device called *prover* is in a legitimate state. It can, for example, be used in critical sensor networks for medical emergency [KLC$^+$10] or wildfire detection [HHSH06, SL16], where a compromised or faulty node could report false information or disrupt operation. A typical RA mechanism is depicted in Figure 1.4. It uses a challenge-response protocol, where the verifier sends a challenge to the prover. The prover will then execute an attestation routine and send back a response based on the challenge and its internal state, for example a hash of its memory. The verifier knows the expected internal state and expects a valid response within a certain timeframe. The attestation routine has to be very carefully implemented to avoid an attacker from forging a valid response or avoiding detection. This can be achieved using special tamper-resistant *hardware* like trusted platform modules (TPMs), carefully written *software*, or *hybrid* approaches that make some assumptions on available hardware. More information can be found in [SL16].

1.4 Related Work

The configuration of real-time safety-critical systems is a very broad topic that includes many research areas such as real-time operating systems, real-time networks, software engineering and reliable hardware design. Security is a related area that is increasingly also looked at in this context, as security breaches may impair safety. In the literature study for this thesis, we focus on optimization problems in real-time networks, especially those using TSN with the TAS mechanism, and the security mechanisms that have been considered.

The recent and extensive survey [SOLM22] provides a good overview of the research in this area. Interesting optimization problems have arisen, for example, in the areas of task and message scheduling, task mapping and message routing. The survey shows that, since the publication of the first papers in 2016, the number of publications in this area has steadily increased (with an exception in 2019). Other surveys [SHM$^+$21] and [DXL$^+$23] have a wider scope and consider other design steps, scheduling, and queuing mechanisms, at the expense of detail. The paper by Steiner [Ste10] is widely considered a seminal work, which inspired many authors. It predates TSN, but it was one of the first to tackle the per-flow scheduling problem in Ethernet networks. In general, a lot of the TSN research evolved from earlier work on other real-time networking technologies like FlexRay or TTEthernet. [PPEP06, TSPS15, CSO16]

Scheduling streams in TSN networks is an NP-hard problem, since the decision problem whether a schedule is feasible is NP-complete, as it can be reduced to the bin-packing problem [Ste10, CSO16, SOLM22]. There are two categories of approaches to solve the scheduling problem: *exact* approaches that find an optimal solution and *heuristic* approaches that find a reasonable solution quickly.

Exact approaches in the literature include Satisfiability Modulo Theories (SMT) [Ste10, CSCS16], Integer Linear Programming (ILP) [PLCS16, VHB$^+$20] and Constraint Programming (CP) [DWZ21, BZP20]. In SMT, scheduling problems are expressed as first-order logic formulas. The SMT solver searches for an assignment of variables which satisfies the formula.[1] ILP describes the scheduling problem as a collection of linear inequalities between integer variables. The solver returns an assignment of integer variables which minimizes a cost function. CP is a more general solution, in which the solver employs techniques like backtracking and constraint propagation to find a solution for a problem declared as a set of variables with domains, constraints on these variables and an optional cost function.

Heuristic approaches include Tabu search [DN16, HGF$^+$20], Simulated Annealing

[1] If an optimal solution according to a cost function is needed, Optimization Modulo Theories (OMT) can be used.

(SA) [BAH+22], Greedy Randomized Adaptive Search Procedures (GRASP) [GP18, GZRP18], Genetic Algorithms (GA) [PSS19, PO18, MJHPC22] and List Scheduling [PTO19]. The simplest heuristic is List Scheduling, in which streams are ordered in a list, e.g., by priority or deadline, and scheduled one-after-another subject to resource constraints. A method for stream placement here could be As-Soon-As-Possible (ASAP) or As-Late-As-Possible (ALAP). The other four methods are metaheuristics: Their goal is to avoid getting stuck in local optima, which is an issue when using local search when only improving steps from an initial solution are taken. GRASP repeatedly selects a solution generated by a greedy randomized algorithm and improves it by a local search. Tabu search follows a local search, but keeps a list of previously selected solutions which become forbidden. To avoid local optima, it allows worsening steps, if no other steps are available. SA follows a local search, but allows worsening steps to be taken with a certain probability, which decreases according to a so-called temperature function. GA is inspired by biology. With GA, there is a pool of candidate solutions and new solutions can be constructed by combining two or more existing solutions with random mutations. The solutions disappear from the pool with a certain probability, which is determined by the cost function. The best solution after a certain number of generations is returned.

These scheduling problems have often been extended with other considerations. One extension is to consider the scheduling of tasks alongside messages, as distributed real-time systems often have complex dependencies between task executions and message transmission [FY21, BZP20]. Other extensions also consider the mapping of tasks, i.e., the assignment of tasks to end-systems, to meet timing, computing resource or energy constraints [Sin06, GBI21, SP18]. Some works consider reliability measures such as message replication [FCD22, FDCL22] and redundant routing [AHM20, GZPS17].

Finally, routing of messages has been extensively covered [GHKS98, WH00]. Multiple authors have also looked at the combined routing and scheduling problem. The authors in [SDT+17, NDR18b] provide ILP solutions and show that optimizing routes increases schedulability compared to fixed routes, and that solving time is more influenced by stream amount than topology size. In [TSPS15] the authors used a Tabu search and in [LPS16] a GRASP metaheuristic to solve similar problems. While some works do a joint routing and scheduling optimization, others do routing and scheduling in separate steps. The authors in [SGRT17] show that the 2-step approach decreases the solution quality for large problems at the benefit of shorter solving times. In [PTO19] the authors presented a heuristic for a more complex application model that allows multicast streams. They were able to solve problems that were infeasible to solve using ILP or separate routing and scheduling. In [PD12] the authors provide a simple set of constraints to solve a general multicast routing problem using constraint programming, which [GZPS17] builds on that to solve a combined topology and route synthesis problem.

Recently, authors have started to present security-aware scheduling problems. A good

overview of potential attacks on TSN networks is presented in [EBN+21]. One desirable security property is confidentiality, such that a man-in-the-middle cannot understand the contents of a message. In [JEP12] the authors present a security-aware task and message scheduling problem, in which messages can be encrypted to provide confidentiality. Each encryption requires an extra task and can either be done slowly in software or faster on special hardware units, the amount of which they try to minimize. The authors in [MAS+19] solve a similar problem, where they try to maximize the security level of applications, by using stronger encryption, which comes at the cost of increasing time overhead. In [AEP18] the authors provide an efficient algorithm that can solve a similar problem online, i.e., at runtime. The authors in [MYPB14] present a threat model in which information from critical tasks can leak to attacker-controlled non-critical tasks through the use of a shared cache. They propose a scheduling algorithm that avoids placing non-critical tasks after critical tasks as much as possible, to avoid time-intensive cache flushes.

A second important property is integrity/authentication, such that a man-in-the-middle cannot modify a message unnoticed. In [MAS+19] the authors solve a combined routing and scheduling problem for TSN with authentication using symmetric block ciphers. The authors in [ZQLY19] provided a task and message scheduling formulation for TTEthernet using the asymmetric TESLA protocol for authentication. Another important property is privacy. In the context of task mapping or message routing, there may be a measure of trust given to each end-system or network link. One work that considers this is [SAR+17], in which the problem is to map tasks with different privacy requirements to edge and cloud resources, where edge resources are more trusted and have shorter response times.

To summarize, there are many promising research directions in the area of safety-critical real-time systems. Since the analysis and scheduling of messages on TSN has been extensively researched, many researchers have extended these problems with other aspects such as task mapping, routing, fault-tolerance and security. Exact solutions are now often complemented by fast and efficient heuristics that enable the configuration of big networks and live reconfiguration. Initial work has been done on security, and we expect this area of research to grow as increasingly interconnected safety-critical systems enter the market.

1.5 Thesis Overview and Contributions

The thesis includes the following three papers:

- **Paper A:** Reusch, N., Barzegaran, M., Zhao, L., Craciunas, S.S. and Pop, P., 2023. Configuration Optimization for Heterogeneous Time-Sensitive Networks. Submitted to *Real-Time Systems Journal, special issue with invited papers from the 30th International Conference on Real-Time Networks and Systems*.
  This paper is an extended version of: Barzegaran, M., Reusch, N., Zhao, L., Craciunas, S.S. and Pop, P., 2022, June. Real-Time Traffic Guarantees in Heterogeneous Time-Sensitive Networks. In *Proceedings of the 30th International Conference on Real-Time Networks and Systems (RTNS) (pp. 46-57).* [BRZ+22]
- **Paper B:** Reusch, N., Craciunas, S.S. and Pop, P., 2022. Dependability-aware routing and scheduling for Time-Sensitive Networking. *IET Cyber-Physical Systems: Theory & Applications*. [RCP22]
  This paper is an extended version of: Reusch, N., Pop, P. and Craciunas, S.S., 2020, December. Safe and secure configuration synthesis for TSN using constraint programming. In *2020 IEEE Real-Time Systems Symposium (RTSS) (pp. 387-390).* [RPC20]
- **Paper C:** Reusch, N. and Pop, P., 2023. Mapping and Scheduling Real-Time Applications on Edge Computing Platforms with Remote Attestation for Security. To be submitted to ACM Transactions on Design Automation of Electronic Systems.
  This paper is an extended version of: Reusch, N. and Pop, P., 2021. Scheduling Real-Time Applications on Edge Computing Platforms with Remote Attestation for Security. In *2021 IEEE/ACM Symposium on Edge Computing (SEC) (pp. 403-408).* [RP21]

Additionally, the following papers, not included in this thesis, were published during the Ph.D studies:

- Reusch, N., Zhao, L., Craciunas, S. S., & Pop, P., 2020. Window-Based Schedule Synthesis for Industrial IEEE 802.1Qbv TSN Networks. *Proceedings of 16Th IEEE International Conference on Factory Communication Systems (WFCS) (pp. 31-34).* [RZCP20]
- Kyriakakis, E., Tange, K., Reusch, N., Zaballa, E. O., Fafoutis, X., Schoeberl, M., & Dragoni, N., 2021. Fault-tolerant Clock Synchronization using Precise Time Protocol Multi-Domain Aggregation. *Proceddings of 2021 IEEE 24th International Symposium on Real-Time Distributed Computing (ISORC) (pp. 114-122).* [KTR+21]

Paper A in Chapter 2 addresses the configuration problem of heterogeneous networks, in which end-systems may not be time synchronized or scheduled. This challenges a fundamental assumption that most of the related work on network scheduling makes. However, removing such an assumption is highly relevant for systems with legacy or off-the-shelf hardware. Without the synchronization of end-systems, we do not know the arrival time of streams on the first switch, and we do not know the order of arrival of multiple streams sharing the same queue. Consequently, we cannot schedule streams/frames individually, but we have to resort to scheduling windows with enough capacity for all streams. To be able to guarantee a streams' deadline, we employ a Network-Calculus-based worst-case delay analysis. I first proposed this window-based scheduling approach in [RZCP20], where we presented a simple heuristic to solve the problem. This work was expanded by Barzegaran et al. in [BRZ$^+$22] with a Constraint Programming formulation, which also allowed the optimization of window offsets, in addition to window lengths. Finally, the paper that is presented in this thesis, is an extension I did of [BRZ$^+$22], which proposes a Simulated Annealing-based metaheuristic. The metaheursitic is compared to the Constraint Programming solution and shown to be much more scalable.

Paper B in Chapter 3 addresses the configuration of networks with high-dependability requirements. In that paper I formulate an extensive combinatorial optimization problem that includes complex applications, composed of tasks with message dependencies, that have real-time but also security and redundancy requirements. It is the first work that proposes the use of the TESLA protocol to provide authentication for security-critical messages in a TSN network. This mandates the execution of additional tasks and the sending of additional messages with their own timing and ordering constraints. I also assume the possibilities of links failing (or being blocked by an attacker), against which we defend by routing streams among multiple disjunct paths. I developed and compared an optimal Constraint Programming and a fast problem-specific Simulated Annealing solution. This paper was first published as work-in-progress in [RPC20], before an extended version with the Simulated Annealing solution was published in [RCP22].

Paper C in Chapter 4 addresses the configuration of Edge Computing platforms and is the first work, that looks at resource management for edge applications while considering real-time applications and security. It is also, to be best of our knowledge, the first work that considers Remote Attestation and its special requirements in a scheduling problem for TSN. Remote Attestation allows checking the integrity of untrusted devices, but is not easy to configure in real-time environments, due to the need for long uninterrupted computations for attestation, which may clash with low period real-time tasks. A special technique is required to securely split the computation into smaller chunks, which I present in the paper. Building on that technique, I formulated an optimization problem and constraints, which minimize the average time between attestations, while guaranteeing real-time constraints of critical applications and minimizing latency for dynamically appearing edge applications. The evaluation remains to be

completed, but we included the evaluation of an earlier version of this work ([RP21]), where we solved a simplified version of this problem with fixed task mapping and routing.

Lastly, we present a test case and scheduling visualization tool in Appendix A. It has been developed to aid the experiments for the above papers, and is available as an open-source project on GitHub[2] and can be tested in the browser[3].

[2]https://github.com/nreusch/TSNConf
[3]https://tsnconf-demo.herokuapp.com/

CHAPTER 2

Paper A: Configuration Optimization for Heterogeneous Time-Sensitive Networks

Time-Sensitive Networking (TSN) collectively defines a set of protocols and standard amendments that enhance IEEE 802.1Q Ethernet nodes with time-aware and fault-tolerant capabilities. Specifically, the IEEE 802.1Qbv amendment defines a timed-gate mechanism that governs the real-time transmission of critical traffic via a so-called Gate Control List (GCL) schedule encoded in each TSN-capable network device. Most TSN scheduling mechanisms are designed for homogeneous TSN networks, in which all network devices must have at least the TSN capabilities related to scheduled gates and time synchronization. However, this assumption is often unrealistic since many distributed applications use heterogeneous TSN networks with legacy or off-the-shelf end systems that are unscheduled and/or unsynchronized. We propose a new scheduling paradigm for heterogeneous TSN networks that intertwines a network calculus worst-case interference analysis within the scheduling step. Through this, we compromise on the solution's optimality to be able to support heterogeneous TSN networks featuring unscheduled and/or unsynchronized end-systems while guaranteeing the real-time properties of critical communication. Within this new paradigm, we propose two solutions to solve the problem, one based on a Constraint Programming formulation and

one based on a Simulated Annealing metaheuristic, that provide different trade-offs and scalability properties. We compare and evaluate our flexible window-based scheduling methods using both synthetic and real-world test cases, validating the correctness and scalability of our implementation. Furthermore, we use OMNET++ to validate the generated GCL schedules.

2.1 Introduction

Standardized communication protocols allowing safety-critical communication with real-time guarantees are becoming increasingly relevant in application domains beyond aerospace, e.g., in industrial automation and even in the automotive sector for advanced driver assistance functions or fully autonomous driving [ALD$^+$21]. Time-Sensitive Networking (TSN) [Ins16c] amends the standard Ethernet protocol with real-time capabilities ranging from clock synchronization and frame preemption to redundancy management and schedule-based traffic shaping [CSCS16]. These novel mechanisms allow standard best-effort (BE) Ethernet traffic to coexist with isolated and guaranteed scheduled traffic (ST) within the same multi-hop switched Ethernet network. The main enablers of this coexistence are a network-wide clock synchronization protocol (802.1ASrev [Ins17a]) defining a global network time, known and bounded device latencies (e.g., switch forwarding delays), and a Time-Aware Shaper (TAS) mechanism [Ins16b] with a global communication schedule implemented in so-called Gate Control Lists (GCLs), facilitating ST traffic with bounded latency and jitter in isolation from BE communication. The TAS mechanism is implemented as a gate for each transmission queue that either allows or denies the sending of frames according to the configured GCL schedule.

Most approaches in the literature that guarantee real-time temporal properties of critical traffic (e.g., [CSCS16, SOCS18, PLCS16]) assume a homogeneous TSN network in which all devices have the time-aware shaper mechanism and are synchronized to a global network time. However, many brownfield deployments in industrial systems require end-to-end guarantees in heterogeneous TSN networks that connect TSN-capable switches with legacy resource-constrained end-points (e.g., PLC, sensors, actuators) that are not easily retrofitted with TSN capabilities. Moreover, in industrial systems that have a long life-cycle and which are dependent on legacy technology [SKJ18], customers are more likely to accept the replacement of switches but not of customized end-points; hence it is more beneficial to transition gradually to new technologies making the integration of legacy systems into TSN networks essential [MAP$^+$21]. Furthermore, converged IT/OT networks in, e.g., fog and edge use-cases [SKJ18], interconnection of TSN networks with, e.g., 5G domains [LLEM$^+$20], or multi-domain TSN networks with different sync mechanisms cannot readily communicate isochronous (fully periodic) traffic [BW21]. Here, the region outside the TSN domain can be viewed as

an unscheduled and unsynchronized end-point sending sporadic critical traffic. Moreover, even if the end-points do have some form of TSN capability (e.g., via switched end-points [vADF+20]), the software layers on top of the TSN hardware mechanism can suffer from non-deterministic jitter and delays, leading to missed transmission slots and ultimately resulting in a sporadic, rather than periodic frame transmission from the end-points

Hence, we investigate in this paper heterogeneous TSN networks where the end-systems are unscheduled and/or unsynchronized (i.e., they do not have the TSN capabilities related to 802.1Qbv and 802.1AS), leading to a sporadic arrival of critical traffic at the TSN-capable switches in the network. Classical schedule generation methods for GCLs enforce either a fully deterministic 0-jitter forwarding of critical frames using either exact SMT/ILP-based solvers [CSCS16] or heuristics [PLCS16], or a more flexible window-based approach that allows some (bounded) degree of interference between critical frames [SOCS18]. However, both methods require that end-systems send the respective critical frames in a scheduled and synchronized way that matches the forwarding schedule defined in the switches, thus requiring TSN capabilities on both end systems and switches. Other work, c.f. [RZCP20, HFG+20], introduce scheduling approaches that do not impose synchronization on the end-systems level but constrain all forwarding GCL windows on switches to be aligned and, furthermore, do not use safe formal verification methods like network calculus for the interference calculation used for the schedule creation.

In this paper, we consider heterogeneous TSN networks, relaxing the requirement that end-systems need to be synchronized and/or scheduled and, furthermore, take into account relative offsets of windows on different nodes. We intertwine the worst-case delay analysis from [ZPGF32] with the scheduling step in order to generate correct schedules where the end-to-end requirements of ST streams (also called flows in previous work) are met. Furthermore, we compare different TSN scheduling approaches that have been proposed in the literature (see Table 2.1 for an overview) to our flexible window-based approach. We define the analysis-driven window optimization problem resulting from our more flexible approach with the goal to be able to enlarge the solution space, reduce computational complexity, and apply it to end-systems without TSN mechanisms. Depending on industrial applications' requirements, our evaluation can help system designers choose the most appropriate combination of configurations for their use-case. The main contributions of the paper are:

- We propose a novel flexible window-based scheduling method that does not individually schedule ST frames and streams, but rather schedules open gate windows for individually scheduled queues. Hence, we can support non-deterministic queue states and thus networks with unscheduled and/or unsynchronized end-systems by integrating the WCD Network Calculus (NC) analysis into the scheduling step. The NC analysis is used to construct a worst-case scenario for each stream to check its schedulability, considering arbitrary arrival times of these streams and the given open

GCL window placements.
- We formulate the "window optimization problem" and provide timing guarantees for real-time streams even in systems with unscheduled and unsynchronized end systems.
- We propose two solutions to solve the problem, one based on a Constraint Programming formulation and one based on a Simulated Annealing metaheuristic.
- For the CP formulation, we propose a proxy function as an alternative to the network calculus analysis in [ZPGF32] and use it to provide timing guarantees inside the CP search.
- We compare and evaluate our flexible window-based scheduling method with existing scheduling methods for TSN networks. The evaluation is based on both synthetic and real-world test cases, validating the correctness and the scalability of our implementation. Furthermore, we use the OMNET++ simulator to validate the generated solutions.

We start with a review of related research, focusing on the existing scheduling mechanisms that we compare our work to, in Section 2.2. We then introduce the system, network, and application models, as well as a description of the main TSN standards, in Section 2.3. We outline the problem in Section 2.4 and present our scheduling optimization strategies based on Constraint Programming (CP) and Simulate Annealing (SA) in Section 2.5 and Section 2.6. We then evaluate and compore both strategies in Section 2.7 before we conclude the paper in Section 2.8.

2.2 Related Work

Scheduling homogeneous TSN networks in which all devices are scheduled and synchronized has been solved in using various heuristics [NDR18a, MAS$^+$18, PO18, PTO19, VBHT22] and optimal ILP- or SMT-based approaches [CSCS16, SOCS18, FDR18, VHT21, ZSEP21b, ZSEP21a]. The most relevant results for providing real-time communication properties in TSN networks, to which we compare our approach, have been presented in [CSCS16, SOCS18, PLCS16, DN16] (summarized in Table 2.1). Originally, the TSN scheduling problem was addressed in [CSCS16] for fully deterministic ST traffic temporal behavior and temporal isolation between ST and non-ST (e.g., AVB, BE) streams/flows, similar to TTEthernet [Ste10, CSO16]. In our comparison, we call this method *0GCL*, since, besides enforcing the required end-to-end latency of ST streams, the scheduling constraints also impose a strictly periodic frame transmission resulting in 0 jitter forwarding of critical traffic. The work in [PLCS16] uses heuristics instead of SMT-solvers to solve the 0-jitter scheduling problem in order to improve scalability while also minimizing the end-to-end latency of AVB streams. In [SOCS18], which we call *Frame-to-Window-based*, the 0-jitter constraint of [CSCS16] is relaxed by allowing more variance in the transmission times

of frames over the hops of their routed paths. This increases the solution space at the expense of increased complexity in the correctness constraints. The method in [SOCS18] can be viewed as window-based scheduling, but, unlike our approach, it requires a unique mapping between GCL windows and frames in order to avoid non-determinism in the queues. In [DN16] the TSN scheduling problem is reduced to having one single queue for ST traffic and solving it using Tabu Search that optimizes the number of guard-bands in order to optimize bandwidth usage.

The main goal of the aforementioned works is similar to ours, namely to allow temporal isolation and compositional system design for ST streams with end-to-end guarantees and deterministic communication behavior. However, all previous methods impose that the end-systems from which the ST traffic originates are synchronized to the rest of the network and have the IEEE 802.1Qbv timed-gate mechanism (i.e., they are scheduled). The open gate windows are then either a result of the frame transmission schedule [CSCS16, PLCS16] or are uniquely associated with predefined subsets of frames [SOCS18]. However, the above property is a significant limitation. In many use cases, especially in the industrial and automotive domains (c.f. [SKJ18]), the end-systems are usually off-the-shelf sensors, microcontrollers, industrial PCs, and edge devices that do not have TSN capabilities.

The work in [RZCP20] proposed a more naive window-based approach (WND) in which the GCL window offsets on different network nodes are not included, thereby essentially limiting the mechanisms by requiring all GCL windows to be lined up between bridges. Moreover, [RZCP20] uses a less advanced analysis step (c.f. [ZPC18]) in the scheduling decisions and a more naive heuristic approach. These limiting assumptions were relaxed in [BRZ$^+$22], which has proposed a Constraint Programming solution to the window optimization problem.

The work in [HGF$^+$20] proposes a scheduling model for TSN networks in industrial automation with different traffic types and a hierarchical scheduling procedure for isochronous traffic. The method proposed in [HFG$^+$20] adopts a so-called stream batching approach, which can be classified as window-based in that it can assign multiple frames to the same GCL window. However, the end-points still need to be synchronized and scheduled, and, additionally, the worst-case delay bounds within the batch windows may lead to deadline misses since they are not based on formal methods like the network calculus framework in our approach. In [SNSH21], the authors present an NC-based analysis for overlapping GCL windows with less pessimistic latency bounds and a scheduling algorithm (FWOS) that focuses on maximizing the allowable overlap of GCL windows to increase the bandwidth of unscheduled traffic without jeopardizing the schedulability of ST traffic. As opposed to our method, [SNSH21] cannot guarantee the schedulability of traffic arriving from unscheduled or unsynchronized end-systems.

Classical approaches like strict priority (SP) and AVB [Ins11] do not require a time-gate mechanism and also work with unscheduled end-systems. In order to provide response-

time guarantees, a worst-case end-to-end timing analysis through methods like network calculus [SHHS03, DAB14] or Compositional Performance Analysis (CPA) [DTE12] are used. In [ZPL$^+$17, ZXZL14, BDNM16], the rate-constrained (RC) streams of TTEthernet [Iss11, SBHP11] are analyzed using network calculus. Other works, such as [WT06a, KM14], study the response-time analysis for TDMA-based networks under the strict priority (SP) and weighted round-robin (WRR) queuing policies. Zhao et al. [ZPGF32] present a worst-case delay analysis, which we use in this paper, for determining the interference delay between ST traffic on the level of flexible GCL windows. Using SP only or leaving all ST windows open for the entire hyperperiod duration (which amounts to SP for ST traffic) will not result in the same response-time bounds and schedulability as our method. Our method can delay specific high-priority ST streams when needed to allow a timely transmission of lower-priority ST streams with a much tighter deadline. Unlike SP, our method uses the IEEE 802.1Qbv timed gates to open and close queues as needed to enforce isolation between traffic classes. With pure SP (or when leaving all gates open at all times), misbehaving end-systems (e.g., babbling-idiot failures) will disrupt all (lower-priority) traffic classes, potentially leading to a loss of all real-time properties of the network.

In [VHB$^+$20], the authors present hardware enhancements to standard IEEE 802.1Qbv bridges (along with correctness constraints for the schedule generation) that remove the need for the isolation constraints between frames scheduled in the same egress queue defined in [CSCS16]. Another hardware adaptation for TSN bridges, which has been proposed by Heilmann et al. [HF19] is to increase the number of non-critical queues in order to improve the bandwidth utilization without impacting the guarantees for critical messages.

2.2.1 In-depth Formal Comparison

In this section, we compare *FWND* with the related work in terms of the objectives and constraints. The related work on ST scheduling using 802.1Qbv consists of: (i) zero-jitter GCL (0GCL) in [CSCS16, PLCS16], (ii) Frame-to-Window-based GCL (FGCL) in [SOCS18], and (iii) Window-based GCL (WND) in [RZCP20].

We summarize the requirements of the ST scheduling approaches from the related work and our *FWND* approach in the first column of Table 2.1. The first three requirements refer to the device capabilities needed for the different approaches, and the next seven rows summarize which constraints and isolation requirements are needed by which approach. The last two rows present the requirements of the complexity of the optimization problem that needs to be solved to provide a solution for the respective approach.

To better understand the fundamental differences and the similarities (in terms of the imposed correctness constraints and schedulability parameters) between our work and

Table 2.1: Scheduling Approaches in TSN.

Requirements	0GCL [CSCS16] [PLCS16]	FGCL [SOCS18]	WND [RZCP20]	FWND
Device Capabilities	802.1Qbv	802.1Qbv	802.1Qbv	802.1Qbv
ES Capabilities	scheduled	scheduled	non-scheduled	non-scheduled
SW Capabilities	scheduled	scheduled	scheduled	scheduled
Frame Constraint	Yes	Yes	Yes	Yes
Link Constraint[1]	Yes	Yes	No	No
Bandwidth Constraint[2]	Implicit	Yes	Yes	Yes
Stream Transmission Constraint[3]	Yes	Yes	No	No
Frame-to-Window Assignment	Implicit	Yes	No	No
Stream/Frame Isolation	Yes	Yes	No	No
End-to-end Constraint	Yes	Yes	Yes	Yes
Schedule synthesis	Yes (intractable)	Yes (intractable)	No (only windows)	No (only windows)
Timing analysis required	No	No	Yes	Yes

[1] This constraint refers to windows on different queues of the same port not allowing to overlap in the time domain. This constraint is called "ordered window constraint" in [SOCS18].
[2] This constraint is called "window size constraint" in [SOCS18].
[3] This constraint is called "Stream Constraint" in [SOCS18].

the approaches that require synchronized and scheduled end systems, we briefly reiterate the formal constraints of previous work. We describe, based on [CSCS16, CSO17, SOCS18], the relevant scheduling constraints for creating correct TSN schedules when using frame- and window-based methods. Table 2.1 shows which of these are needed by which approach.

We adapt some notations from [CSCS16, CSO17] to describe the constraints and assume certain simplifications without loss of generality, e.g. the macrotick is the same in all devices, all streams have only one frame per period, the propagation delay d_p is 0. We refer the reader to [CSCS16, SOCS18] for a complete and generalized formal definition of the correctness constraints. We denote the messages (frames) of a stream f_i on a link $[v_a, v_b]$ as $m_i^{[v_a,v_b]}$. A message $m_i^{[v_a,v_b]}$ is defined by the tuple $\langle m_i^{[v_a,v_b]}.\phi, m_i^{[v_a,v_b]}.l \rangle$, denoting the transmission time and duration of the frame on the respective link [CSCS16, CSO17].

Frame Constraint. Any frame belonging to a critical stream has to be transmitted between time 0 and its period T_i. To enforce this, we have the frame constraint from [CSCS16]:

$$\forall f_i \in \mathcal{F}, \forall [v_a, v_b] \in f_i.r : \\ \left(m_i^{[v_a,v_b]}.\phi \geq 0\right) \wedge \left(m_i^{[v_a,v_b]}.\phi \leq f_i.T - m_i^{[v_a,v_b]}.l\right).$$

Link Constraint. A physical constraint of Ethernet-based networks is that only one frame can be on the wire from one port to another at a time. In [CSCS16] and [SOCS18]

this constraint is expressed as windows on two different queues of the same egress port not being able to overlap. In [RZCP20] and the FWND method presented in this paper, windows on different queues may overlap, leading to added interference delays since, naturally, only one frame can be sent on the physical link at a time. Still, in both [RZCP20] and the FWND, solutions where windows overlap are excluded since there is no added improvement from such schedules.

The link constraint adapted from [CSCS16] is hence:

$$\forall [v_a, v_b] \in \boldsymbol{E}, \forall m_i^{[v_a,v_b]}, m_j^{[v_a,v_b]} (i \neq j),$$
$$\forall a \in [0, hp_i^j / f_i.T - 1], \forall b \in [0, hp_i^j / f_j.T - 1]:$$
$$\left(m_i^{[v_a,v_b]}.\phi + a \times f_i.T \geq m_j^{[v_a,v_b]}.\phi + b \times f_j.T + m_j^{[v_a,v_b]}.l\right) \vee$$
$$\left(m_j^{[v_a,v_b]}.\phi + b \times f_j.T \geq m_i^{[v_a,v_b]}.\phi + a \times f_i.T + m_i^{[v_a,v_b]}.l\right),$$

where $hp_i^j = lcm(f_i.T, f_j.T)$ is the hyperperiod of f_i and f_j.

Bandwidth Constraint. The bandwidth constraint expressed explicitly in our method ensures that there is no infinite backlog, i.e., the windows for the streams are large enough that the frames of the streams can be transmitted at some point. In 0GCL from [CSCS16] this constraint is implicit since the schedule is created without the separation of streams and windows, meaning that each window is large enough to transmit the respective frames. In [SOCS18] there is no one-to-one assignment between frames and windows; however, the window size constraint is equivalent to the bandwidth constraint. Using this constraint, the length of the gate open window is required to be equal to the sum of the frame lengths that have been assigned to it. In [RZCP20] the bandwidth constraint is explicit in the conditions for the correctness of the schedule generation.

Stream Transmission Constraint. The stream transmission constraint expresses that the propagation of frames of a stream follows the sequential order along the path of the stream. This (optional) constraint enforces that a frame is forwarded by a device only after it has been received at that device also taking into account the network precision, denoted with δ:

$$\forall f_i \in \mathcal{F}, \forall [v_a, v_x], [v_x, v_b] \in f_i.r, \forall m_i^{[v_a,v_x]}, \forall m_i^{[v_x,v_b]}:$$
$$m_i^{[v_x,v_b]}.\phi - \delta \geq m_i^{[v_a,v_x]}.\phi + m_i^{[v_a,v_x]}.l.$$

In FWND (and also in [RZCP20]) this constraint is not explicitly needed since there is no predefined assignment of frames to windows and hence, there is no explicit ordering needed in sequential hops along the route of a stream, i.e., the transmission GCL window which is used at a certain time will depend on the enqueueing order at that time in the egress queue.

End-to-End Constraint. The maximum end-to-end latency constraint (expressed by the deadline $f_i.D$) enforces a maximum time between the sending and the reception of a stream. We denote the sending link of stream f_i with $src(f_i)$ and the last link before the receiving node with $dest(f_i)$. The maximum end-to-end latency constraint [CSCS16] is hence

$$\forall f_i \in \mathcal{F} : m_i^{dest(f_i)}.\phi + m_i^{dest(f_i)}.L - m_i^{src(f_i)}.\phi \leq f_i.D - \delta.$$

Here again the network precision δ needs to be taken into account since the local times of the sending and receiving devices can deviate by at most δ.

802.1Qbv Stream/Frame Isolation. Due to the non-determinism problem in TSN (c.f. Section 2.3.2), previous solutions (e.g., [CSCS16, SOCS18]) need an isolation constraint that maintains queue determinism. We refer the reader to [CSCS16] for an in-depth explanation, and only summarize here the stream and frame isolation constraint adapted from [CSCS16]. Let $m_i^{[v_a,v_b]}$ and $m_j^{[v_a,v_b]}$ be, respectively, the frame instances of $f_i \in \mathcal{F}$ and $f_j \in \mathcal{F}$ scheduled on the outgoing link $[v_a, v_b]$ of device v_a. Stream f_i arrives at the device v_a from some device v_x on link $[v_x, v_a]$. Similarly, stream f_j arrives from another device v_y on incoming link $[v_y, v_a]$. The simplified stream isolation constraint adapted from [CSCS16], under the assumption that the macrotick of the involved devices is the same, is as follows:

$$\begin{aligned}
&\forall [v_a, v_b] \in \boldsymbol{E}, \forall m_i^{[v_a,v_b]}, m_j^{[v_a,v_b]} (i \neq j),\\
&\forall a \in [0, hp_i^j / f_i.T - 1], \forall b \in [0, hp_i^j / f_j.T - 1] :\\
&\left(m_i^{[v_a,v_b]}.\phi + a \times f_i.T + \delta \leq m_j^{[v_y,v_a]}.\phi + b \times f_j.T\right) \vee\\
&\left(m_j^{[v_a,v_b]}.\phi + b \times f_j.T + \delta \leq m_i^{[v_x,v_a]}.\phi + a \times f_i.T\right).
\end{aligned}$$

Here again $hp_i^j = lcm(f_i.T, f_j.T)$ is the hyperperiod of f_i and f_j. The constraint ensures that once a stream arrives at a device, no other stream can enter the device until the first stream has been sent.

The above constraints apply to frames that are placed in the same queue on the egress port. However, the scheduler may choose (if possible) to place streams in different queues, isolating them in the space domain. Hence, the complete constraint [CSCS16] for frame/stream isolation for two streams f_i and f_j scheduled on the same link $[v_a, v_b]$ can be expressed as

$$\left(\Phi_{[v_a,v_b]}(f_i, f_j)\right) \vee \left(m_i^{[v_a,v_b]}.q \neq m_j^{[v_a,v_b]}.q\right),$$

with $m_i^{[v_a,v_b]}.q \leq N_{tt}$ and $m_j^{[v_a,v_b]}.q \leq N_{tt}$ and where $\Phi_{[v_a,v_b]}(f_i, f_j)$ denotes either the stream or frame isolation constraint from before.

Decoupling of frames. So far, the constraints were applicable on the level of frames, and the open windows of the GCLs were constructed from the resulting frame schedule. The approach in [SOCS18] decouples the frame transmission from the respective open gate windows defined in the GCLs, similar to our approach[1]. However, in [SOCS18] the requirement is that there is a unique assignment of which frames are transmitted in which windows, although also multiple frames can be assigned to be sent in the same window. Hence, the assignment of frames and, consequently, the length of each gate open window are, therefore, an output of the scheduler. Therefore, we have to construct additional constraints when (partially) decoupling frames from windows. For a more in-depth description and formalization of these constraints, we refer the reader to [SOCS18].

Frame-to-Window Assignment. The frame-to-window assignment restricts a frame to be assigned to a specific window, although multiple frames can be assigned to the same window. In [CSCS16] each frame is assigned implicitly to exactly one GCL window.

Comparing the existing approaches with the one proposed in this paper, we see that the choice of scheduling mechanism is, on the one hand, highly use-case specific and, on the other hand, is constrained by the available TSN hardware capabilities in the network nodes. While the frame- and window-based methods from related work result in precise schedules that emulate either a 0- or constrained-jitter approach (e.g., like in TTEthernet), they require end systems to not only be synchronized to the network time but also the end devices to have 802.1Qbv capabilities, i.e., to be scheduled. This limitation might be too restrictive for many real-world systems relying on off-the-shelf sensors, processing, and actuating nodes. While our *FWND* method overcomes this limitation, it does require a worst-case end-to-end analysis that introduces a level of pessimism into the timing bounds, thereby reducing the schedulability space for some use cases. However, as seen in Table 2.1, our method does not require many of the constraints imposed on the streams and scheduled devices from previous work, thereby reducing the complexity of the schedule synthesis.

[1] Note that [CSCS16, SOCS18, PLCS16] cannot be used in our context because they require scheduled and synchronized ESs.

Table 2.2: Summary of System Model Notations.

Symbol	System model
$G = (\boldsymbol{V}, \boldsymbol{E})$	Network graph with nodes ($\boldsymbol{V}$) and links ($\boldsymbol{E}$)
$[v_a, v_b] \in \boldsymbol{E}$	Link
$[v_a, v_b].C$,	Link speed
$[v_a, v_b].mt$	Link macrotick
$p \in P$	Output port
$p.Q$	Eight priority queues in an output port p
$q \in p.Q_{ST}$	A queue used for ST traffic in p
$\langle \phi, w, T \rangle_q$	GCL configuration for a queue $q \in p.Q_{ST}$, where $q.\phi$, $q.w$, and $q.T$ are the window offset, length, and period for queue q, respectively.
$f.l, f.T$	Payload size and period of a stream $f \in \mathcal{F}$
$f.P, f.D$	Priority, and deadline of a stream $f \in \mathcal{F}$
$f.r$	Route for a stream $f \in \mathcal{F}$

2.3 System Model

This section defines our system model for which we summarize the notation in Table 2.2.

2.3.1 Network Model

We represent the network as a directed graph $G = (\boldsymbol{V}, \boldsymbol{E})$ where $\boldsymbol{V} = \boldsymbol{ES} \bigcup \boldsymbol{SW}$ is the set of end systems (ES) and switches (SW) (also called nodes), and $\boldsymbol{E}$ is the set of bi-directional full-duplex physical links. An ES can receive and send network traffic while SWs are forwarding nodes through which the traffic is routed. The edges $\boldsymbol{E}$ of the graph represent the full-duplex physical links between two nodes, $\boldsymbol{E} \subseteq \boldsymbol{V} \times \boldsymbol{V}$. If there is a physical link between two nodes $v_a, v_b \in \boldsymbol{V}$, then there exist two ordered tuples $[v_a, v_b], [v_b, v_a] \in \boldsymbol{E}$. An equivalence between output ports $p \in P$ and links $[v_a, v_b] \in \boldsymbol{E}$ can be drawn, as each output port is connected to exactly one link. A link $[v_a, v_b] \in \boldsymbol{E}$ is defined by the link speed C (Mbps), propagation delay d_p (which is a function of the physical medium and the link length), and the macrotick mt. The macrotick is the length of a discrete time unit in the network, defining the granularity of the scheduling timeline [CSCS16]. Without loss of generality, we assume $d_p = 0$ in this paper.

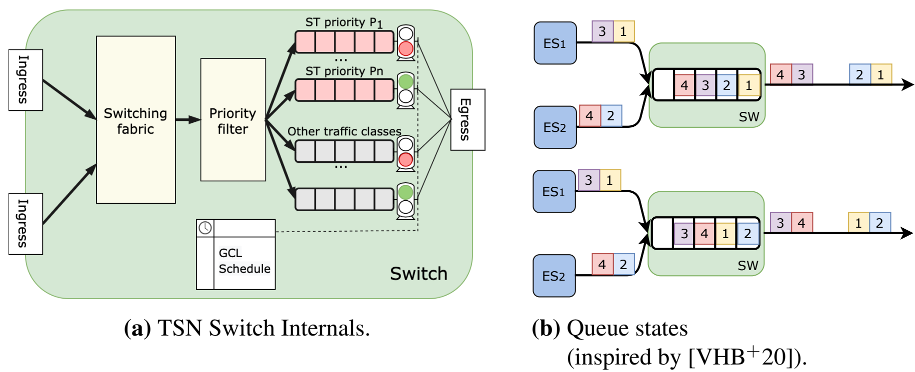
As opposed to previous work, we do not require that end-system are either synchronized or scheduled. Since ESs can be unsynchronized and unscheduled, they transmit

frames according to a strict priority (SP) mechanism. Switches still need to be synchronized and scheduled using the 802.1ASrev and 802.1Qbv, respectively.

2.3.2 Switch Model

Figure 2.1a depicts the internals of a TSN switch. The switching fabric decides, based on the internal routing table to which output port p a received frame will be forwarded. Each egress port has a priority filter that determines in which of the available 8 queues/traffic-classes $q \in p.Q$ of that port a frame will be put. Within a queue, frames are transmitted in first-in-first-out (FIFO) order. Similar to [CSCS16], a subset ($p.Q_{ST}$) of the queues are reserved for ST traffic, while the rest ($p.\overline{Q}$) are used for non-critical communication. As opposed to regular 802.1Q bridges, where enqueued frames are sent out according to their respective priority, in 802.1Qbv bridges, there is a Time-Aware Shaper (TAS), also called timed-gate, associated with each queue and positioned behind it. A timed-gate can be either in an *open* (O) or *closed* (C) state. When the gate is open, traffic from the respective queue is allowed to be transmitted, while a closed gate will not allow transmission, even if the queue is not empty. When multiple gates are open simultaneously, the highest priority queue has preference, blocking others until it is empty or the corresponding gate is closed. The 802.1Qbv standard includes a mechanism to ensure that no frames can be transmitted beyond the respective gate's closing point. This look-ahead checks whether the entire frame present in the queue can be fully transmitted before the gate closes and, if not, it will not start the transmission.

The state of the queues is encoded in a GCL, which acts on the level of traffic-classes (contrary to, e.g., TTEthernet [Iss11]) instead of on an individual frame level [CSO17]. Hence, an imperfect time synchronization, frame loss, ingress policing (c.f. [CSCS16]),



(a) TSN Switch Internals.

(b) Queue states (inspired by [VHB+20]).

Figure 2.1: TSN Switch model and Queue Interference.

or the variance in the arrival of frames from unscheduled and/or unsynchronized ESs may lead to non-determinism in the state of the egress queues and, as a consequence, in the whole network. If the state of the queue is not deterministic at runtime, the order and timing of the sending of ST frames can vary dynamically. In Figure 2.1b, the schedule for the queue of the (simplified) switch SW, opens for two frames and then, sometime later, for the duration of another two frames. The arrival of frames from unscheduled and/or unsynchronized end systems may lead to a different pattern in the egress queue of the switch, as illustrated in the top and bottom figures of Figure 2.1b. Note that we do not actually know the arrival times of the frames, and what we depict in the figure are just two scenarios to illustrate the non-determinism. There may be scenarios where one of the frames, e.g., frame “2”, arrives much later. This variance makes it impossible to isolate frames in windows and obtain deterministic queue states, and, as a consequence, deterministic egress transmission patterns, as required by previous methods for TSN scheduling (e.g. [CSCS16, SOCS18, PLCS16, DN16]). We refer the reader to [CSCS16] for an in-depth explanation of the TSN non-determinism problem.

The queue configuration is expressed by $q = \langle Q_{ST}, \overline{Q} \rangle$. The decision in which queue to place frames is taken either according to the priority code point (PCP) of the VLAN tag or according to the priority assignment of the IEEE 802.1Qci mechanism. In order to formulate the scheduling problem, the GCL configuration is defined as a tuple $\langle \phi, w, T \rangle_q$ for each queue $q \in p.Q_{ST}$ in an output port p, with the window offset ϕ, window length w and window period T.

2.3.3 Application Model

The traffic class we focus on in this paper is scheduled traffic (ST), also called time-sensitive traffic. ST traffic is defined as having requirements on the bounded end-to-end latency and/or minimal jitter [CSCS16]. Communication requirements of ST traffic itself are modeled with the concept of streams (also called flows), representing a communication from one sender (talker) to one or multiple receivers (listeners). We define the set of ST streams in the network as $\mathcal{F}$. A stream $f \in \mathcal{F}$ is expressed as the tuple $\langle l, T, P, D \rangle_f$, including the frame size, the stream period in the source ES, the priority of the stream, and the required deadline representing the upper bound on the end-to-end delay of the stream.

The route for each stream is statically defined as an ordered sequence of directed links, e.g., a stream $f \in F$ sending from a source ES v_1 to another destination ES v_n has the route $r = [[v_1, v_2], ..., [v_{n-1}, v_n]]$. Without loss of generality, the notation is simplified by limiting the number of destination ES to one, i.e., unicast communication. Please note that the model can be easily extended to multicast communication by adding each sender-receiver pair as a stand-alone stream with additional constraints between them on the common path.

We assume that streams arrive sporadically, meaning at a random time, but with a minimum interarrival time of the stream's period.

2.4 Problem Formulation

Given (1) a set of streams $\mathcal{F}$ with statically defined routes $\mathcal{R}$, and (2) a network graph G, we are interested in determining GCLs, which is equivalent to determining (i) the offset of windows $q.\phi$, (ii) the length of windows $q.w$, and (iii) the period of windows $q.T$ such that the deadlines of all streams are satisfied and the overall bandwidth utilization (c.f. Section 2.5.1) is minimized.

We remind the reader that with flexible window-based scheduling we do not know the arrival times of frames, and frames of different ST streams may interfere with each other. Frames that arrive earlier will delay frames that arrive later; also, a frame may need to wait until a gate is open, or arrive at a time just before a gate closure and cannot fit in the interval that remains for transmission.

Once the problem is unschedulable (some stream deadlines are missed), we determine the solution in which the number of missed stream deadlines, are minimized. In this paper, we use Network Calculus [JYP01] to calculate the worst-case delay of streams.

Our problem is intractable, i.e., the decision problem associated with the scheduling problem has been proved to be (NP)-complete in the strong sense [Sin07a]. We present two solutions for this problem. Our first solution is based on Constraint Programming (CP), see Section 2.5. CP is an exact mathematical programming approach that attempts to find an optimal solution. However, as our experiments in Section 2.7 will show, CP cannot handle realistic test cases. Hence, we also propose a second solution, based on a Simulated Annealing (SA) metaheuristic, see Section 2.6. Metaheuristics have been used as an alternative to exact optimization methods such as CP [BK^{+}05].

2.4.1 Motivational Example

Let us illustrate the importance of determining optimized windows. Recall that with flexible window-based scheduling we do not know the arrival times of frames, and frames of different ST streams may interfere with each other. Frames that arrive earlier will delay frames that arrive later; also, a frame may need to wait until a gate is open, or arrive at a time just before a gate closure and cannot fit in the interval that remains for transmission.

We illustrate in Figure 2.2 three window configurations (a), (b), and (c), motivating the need to optimize the windows. The vertical axis represents each egress port in the network, and the horizontal axis represents the timeline. The tall grey rectangles give the gate open time for a priority queue. As mentioned, we do not know the arrival times of the frames; thus it is necessary to provide a formal analysis method to ensure real-time performance. In this paper, we use the Network Calculus (NC)-based approach from [ZPGF32] to determine the worst-case end-to-end delay bounds (WCDs) for each stream. In the motivational example, the WCDs are determined by constructing a worst-case scenario for each stream. Hence, in Figure 2.2 we show worst-case scenarios. The red and blue rectangles in the figure represent ST frames' transmission. There are two periodic streams f_1 (blue rectangles) and f_2 (red rectangles) with the same frame size and priority. We use the arrows pointing down to mark the arrival time for ST frames creating the worst-case case for f_2. Let us assume that the deadline of each stream equals its period $f_i.T$. In each configuration in Figure 2.2 we show that arrival scenario which would lead to the worst-case situation for frame f_2, i.e., the largest WCD for f_2.

Since the ESs are unscheduled (without TAS) and/or unsynchronized, the frames can arrive and be transmitted by the ES at any time (the offset of a periodic stream on the ES is in an arbitrary relationship with the offsets of the windows in the SWs). However, on the switches, frame transmissions are allowed to be forwarded only during the scheduled window. The worst-case for f_2 happens when the frame (1.1) of f_1 arrives on $[ES_1, SW_1]$ slightly earlier than the frame (2.1) of f_2 arrives on $[ES_2, SW_1]$, and at the same time, they arrive on the subsequent egress port $[SW_1, SW_2]$ at a time when the

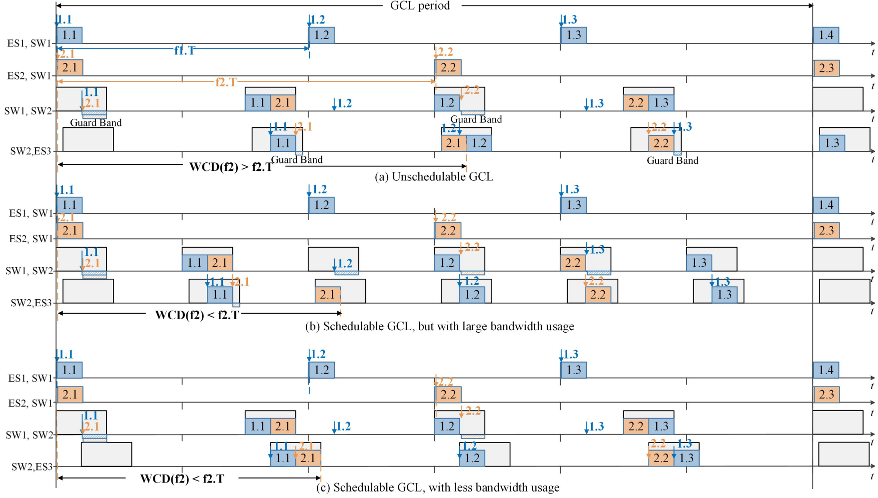


Figure 2.2: Example Window Configurations.

remaining time during the current window is smaller than the frame transmission time. In this case, the guard band delays the frames until the next window slot.

With the windows configuration in Figure 2.2(a), the WCD of f_2 is larger than its deadline, i.e., $WCD(f_2) > f_2.T$, hence, f_2 is not schedulable. If the window period is narrowed down, as shown in Figure 2.2(b), the WCD of f_2 satisfies its deadline. However, there is a large bandwidth usage occupied by the windows. Figure 2.2(c) uses the same window period as in Figure 2.2(a) but changes the window offset. As can be seen in the figure, the WCD of the stream f_2 is also smaller than its deadline $f_2.T$, and compared with Figure 2.2(b), the bandwidth usage of the windows is reduced. With the increasing complexity of the network, e.g., multiple streams joining and leaving at any switch in the network and/or an increased number of ST streams and priorities, an optimized window configuration cannot be done manually; therefore, optimization algorithms are needed to solve this problem.

2.5 Constraint Programming Window Optimization

In this section, we present a solution based on a Constraint Programming formulation. Although CP can perform an exhaustive search and find the optimal solution, this is infeasible for large networks. Hence, we propose a strategy called Constraint Programming-based Window Optimization (CPWO) that is able to "prune" the search to find optimized solutions in a reasonable time, at the expense of optimality. CPWO has two features intended to speed up the search:

(i) A metaheuristic search traversal strategy: CP solvers can be configured with user-defined search strategies, which enforce a custom order for selecting variables for assignment and for selecting the values from the variable's domain. Here, we use a *metaheuristic* strategy based on Tabu Search [BK+05].

(ii) A timing constraint specified in the CP model that prunes the search space: Ideally, the WCD Analysis would be called for each new solution. However, an NC-based analysis is time-consuming, and it would slow down the search considerably if called each time the CP solver visits a new valid solution. Hence, we have introduced "search pruning" constraints in the CP model (the "Timing (pruning)" constraints in the "CP model" box in Figure 2.3), explained in Section 2.5.4.

CPWO takes as the inputs the architecture and application models and outputs a set of the best solutions found during search (see Figure 2.3). We use CP to search for solutions (the "CP solver" box). CP performs a systematic search to assign the values of variables to satisfy a set of constraints and optimize an objective function, see the "CP model" box: the sets of variables are defined in Section 2.5.2, the constraints in

Section 2.5.3 and the objective function in Section 2.5.1. A feasible solution is a valid solution that is schedulable, i.e., the worst-case delays (WCDs) of streams are within their deadlines. Since it is impractical to check for schedulability within a CP formulation, we employ instead the Network Calculus (NC)-based approach from [ZPGF32] to determine the WCDs, see the "WCD Analysis" box in Figure 2.3. The WCD Analysis is called every time the CP solver finds a "new solution" which is valid with respect to the CP constraints. The "new solution" is not schedulable if the calculated latency upper bounds are larger than the deadlines of some critical streams.

These timing constraints implement a crude analysis that indicates if a solution may be schedulable and are solely used by the CP solver to eliminate solutions from the search space. These constraints may lead to both "relaxed-pruning" scenarios that are actually unschedulable or "aggressive-pruning" scenarios that eliminate solutions that are schedulable. The proxy function (pruning constraint) can thus be parameterized to trade-off runtime performance for search-space pruning in the CP-model.

The timing constraints assume that for a given stream, its frames in a queue will be delayed by other frames in the same queue, including a backlog of frames of the same stream. A parameter $\mathcal{B}$ is used to adjust the number of frames in the backlog, tuning the pruning level of the CP model's timing constraints. Note that NC still checks the actual schedulability, so it does not matter if the CP analysis is too relaxed—this will only prune fewer solutions, slowing down the search. However, using overly aggressive pruning runs the risk of eliminating schedulable solutions of good quality. We consider that $\mathcal{B}$ is given by the user, controlling how fast to explore the search space. In the experiments, we adjusted $\mathcal{B}$ based on the feedback from the WCD Analysis and the pruning constraint. If, during a CPWO run, the pruning constraint from Section 2.5.4 was invoked too often, we decreased $\mathcal{B}$, as it was pruning too aggressively; otherwise, if the WCD analysis was invoked too often and was reporting that the solutions were schedulable, we increased $\mathcal{B}$.

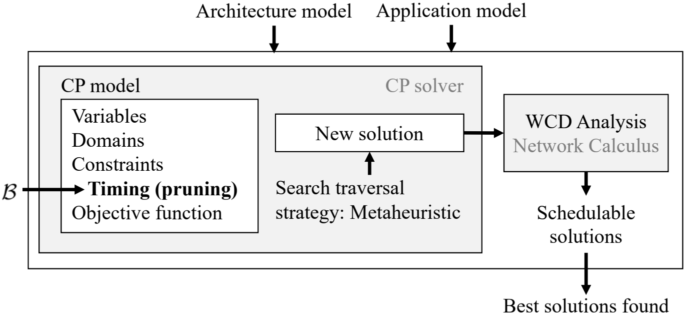


Figure 2.3: Overview of our CPWO Optimization Strategy.

Table 2.3: Definition of Terms for CP Model Formulation.

Term	Definition
$\mathcal{N}(P)$	Total number of windows assigned to priority queues
$\mathcal{K}(p)$	Hyperperiod of the port p
$\mathcal{L}(q)$	Maximum size of any frame from all streams assigned to q
$\mathcal{GB}(q)$	Maximum transmission time of ST frames competing in q
$\mathcal{R}(q)$	All streams assigned to the queue q
$\mathcal{X}(q)$	All streams arriving from a switch and assigned to the queue q

We first define the terms needed for the CP model in Table 2.3. Then, we continue with the definition of the objective function, model variables, and constraints of the CP model.

2.5.1 CP Objective Function

The CP solver uses the objective function Ω, which minimizes the average bandwidth usage:

$$\forall p \in P, \forall q \in p.Q : \Omega = \frac{\sum \frac{q.w}{q.T}}{\mathcal{N}(P)}. \tag{2.1}$$

The average bandwidth usage is calculated as the sum of each window's utilization, i.e., the window length over its period, divided by the total number of windows in the CP model. Note that solutions found by a CP solver are guaranteed to satisfy the constraints defined in Section 2.5.3. In addition, the schedulability is checked with the NC-based WCD Analysis [ZPGF32].

2.5.2 Variables

The model variables are the offset, length, and period of each window, see Section 2.3.2. For each variable, we define a domain, which is a set of finite values that can be assigned to the variable. CP decides the values of the variables as an integer from their domain in each visited solution during the search. The domains of offset $q.\phi$, length

$q.w$, and period $q.T$ variables are defined, respectively, by

$$\begin{aligned}&\forall p \in P, \forall q \in p.Q:\\&0 < q.T \leq \frac{\mathcal{K}(p)}{[v_a, v_b].mt}, \quad 0 \leq q.\phi \leq \frac{\mathcal{K}(p)}{[v_a, v_b].mt},\\&\frac{\mathcal{L}(q)}{[v_a, v_b].mt \times [v_a, v_b].C} + \mathcal{GB}(q) \leq q.w \leq \frac{\mathcal{K}(p)}{[v_a, v_b].mt}.\end{aligned} \tag{2.2}$$

The domain of the window period is defined in the range from 0 to the hyperperiod of the respective port p, i.e., the Least Common Multiple (LCM) of all the stream periods forwarded via the port. The window period is an integer and cannot be zero. The domain of the window offset is defined in the range from 0 to the hyperperiod of the respective port p. Finally, the domain of the window length is defined in the range from minimum accepted window length to the hyperperiod of the respective port p. The minimum accepted window length is the length required to transfer the largest frame from all streams assigned to the queue q, protected by the guard band $\mathcal{GB}(q)$ of the queue. A port p is attached to only one link $[v_a, v_b]$; and values and domains are scaled by the macrotick *mt* of the respective link.

2.5.3 Constraints

The first three constraints need to be satisfied by a valid solution: (1) the window is valid, (2) two windows in the same port do not overlap, and (3) windows' bandwidth is not exceeded. The last two constraints reduce the search space by restricting the periods of (4) queues and (5) windows to harmonic values in relation to the hyperperiod. Harmonicity may eliminate some feasible solutions, but we use this heuristic strategy to speed up the search.

(1) The **Window Validity Constraint** (Equation 2.3) states that the offset plus the length of a window should be smaller or equal to the window's period:

$$\forall p \in P, \forall q \in p.Q: \quad (q.w + q.\phi) \leq q.T. \tag{2.3}$$

(2) **Non-overlapping Constraint** (Equation 2.4). Since we search for solutions in which windows of the same port do not overlap, the opening or closing of each window on the same port (defined by its offset and the sum of its offset and length, respectively) is not in the range of another window, over all period instances:

$$\begin{aligned}&\forall p \in P, \forall q \in p.Q, \forall q' \in p.Q, T_{q,q'} = max(q.T, q'T),\\&\forall a \in [0, T_{q,q'}/q.T), \forall b \in [0, T_{q,q'}/q'.T):\\&(q.\phi + q.w + a \times q.T) \leq (q'.\phi + b \times q'.T) \vee\\&(q'.\phi + q'.w + b \times q'.T) \leq (q.\phi + a \times q.T)\end{aligned} \tag{2.4}$$

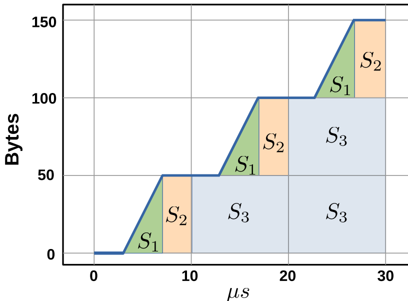


Figure 2.4: Example Capacity for a Window.

(3) The **Bandwidth Constraint** (Eq. (2.5)) ensures that all the windows have enough bandwidth for the assigned streams:

$$\forall p \in P, \forall q \in p.Q, \forall f \in \mathcal{F}(q) : \frac{q.w}{q.T} \geq \sum \frac{f.l}{f.T}. \tag{2.5}$$

where $\mathcal{F}(q)$ is the set of streams assigned to the queue q.

(4) The **Port Period Constraint** (Equation 2.6) imposes that the periods of all the queues in a port should be harmonic. This constraint is used to avoid window overlapping and to reduce the search space.

$$\forall p \in P, \forall q \in p.Q, \forall q' \in p.Q : (q.T \% q'.T = 0) \vee (q'.T \% q.T = 0). \tag{2.6}$$

(5) The **Period Limit Constraint** (Equation 2.7) reduces the search space by considering window periods $q.T$ that are harmonic with the hyperperiod of the port $\mathcal{K}(p)$ (divide it):

$$\forall p \in P, \forall q \in p.Q : \quad \mathcal{K}(p) \% q.T = 0. \tag{2.7}$$

2.5.4 Timing Constraints

As mentioned, it is infeasible to use a Network Calculus-based worst-case delay analysis to check the schedulability of *each* solution visited. Thus, we have defined a *Timing Constraint* as a way to prune the search space. Every solution that is not eliminated via this timing constraint is evaluated for schedulability with the NC WCD analysis. The timing constraint is a heuristic that prunes the search space of (potentially unschedulable) solutions; it is not a sufficient nor a necessary schedulability test. The timing constraint is related to the optimality of the solution, not to its correctness in terms of schedulability. A too aggressive pruning may eliminate good quality solutions, and too

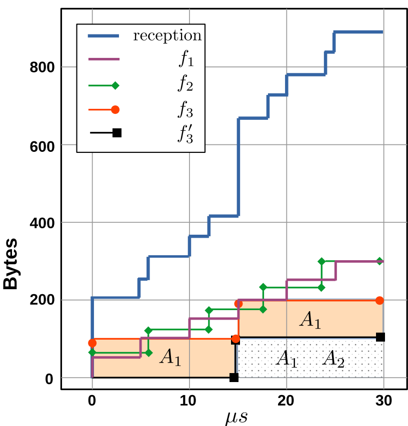


Figure 2.5: Example Capacity and Transmission Demand for a Window.

little pruning will slow down the search because the NC WCD analysis is invoked too often.

The challenge is that the min+ algebra used by NC cannot be directly expressed in first-order formulation of CP. However, the NC formulation from [ZPGF32] has inspired us in defining the CP timing constraints. The *Timing Constraint* is defined in Equation 2.8 and uses the concepts of *window capacity* $\mathcal{W}_C$ and *transmission demand* $\mathcal{W}_D$ to direct the CP solver to visit only those solutions where the *capacity* of each window, i.e., the amount of time available to transmit frames assigned to its queue, is greater than or equal to its transmission *demand*, i.e., the amount of transmission time required by the frames in the queue. A window capacity larger than the transmission demand indicates that a solution has high chances to be schedulable:

$$\forall p \in P, \forall q \in p.Q : \mathcal{W}_D \leq \mathcal{W}_C. \tag{2.8}$$

Thus, we first calculate the capacity $\mathcal{W}_C$ of each window within the hyperperiod. This capacity is similar to the NC concept of a service curve, and its calculation is similar to the service curves proposed in the literature [Wan06] for resources that use Time-Division Multiple Access (TDMA), which is how our windows behave. For e.g., a window with a period of 10 μs, a length of 4 μs, and an offset of 3 μs; forwards 150 bytes over a 100 Mbps link in a hyperperiod of 30 μs. In Figure 2.4, the capacity of such a window is depicted, where the blue line shows the throughput of the window for transferring data. The capacity increases when the window opens (the rising slopes of the curve). The effect of window offset on the capacity (the area under the curve) can be observed in the figure. The function $\mathcal{W}_C$ calculates the area under the curve to characterize the amount of capacity for a window in a hyperperiod, defined in Equation 2.9, where the link $[v_a, v_b]$ is attached to the port p and assigned to the queue q;

and function $\mathcal{Y}$ captures the transmission time of a single byte through link $[v_a, v_b]$.

To calculate the area under the curve, we consider 3 terms that are S_1, S_2, and S_3. They represent the total area under the curve caused by the window length, the window closure in the remainder of the window period, and the window period, respectively. The $\mathcal{W}_C$ value of the example in Figure 2.4 is 2,250 $Bytes \times \mu s$, where the S terms are shown.

$$
\begin{aligned}
&\forall p \in P, \forall q \in p.Q: \\
&I = \frac{\mathcal{K}(p)}{q.T}, \quad J = \frac{(q.w - \mathcal{GB}(q)) \times \mathcal{Y}([v_a, v_b])}{[v_a, v_b].C}, \\
&S_1 = I \times \frac{q.w \times J}{2}, \quad S_2 = I \times (q.T - q.w. - q.\phi) \times J, \\
&S_3 = \frac{I \times (I-1)}{2} \times q.T \times J, \quad \mathcal{W}_C = S_1 + S_2 + S_3
\end{aligned}
\tag{2.9}
$$

Secondly, we calculate the transmission demand $\mathcal{W}_D$ using Equation 2.10, where $\mathcal{R}(q)$ captures all the streams that are assigned to the queue q. The transmission demand is inspired by the *arrival curves* of NC. These are carefully determined in NC considering that the streams pass via switches and may change their arrival patterns [ZPGF32]. In our case, we have made the following simplifying assumptions to be able to express the "transmission demand" in CP. We assume that all streams are strictly periodic and arrive at the beginning of their respective periods. This is "optimistic" with respect to NC in the sense that NC may determine that some of the streams have a bursty behavior when they reach our window. To compensate for this, we consider that those streams that arrive from a switch may be bursty and thus have a backlog $\mathcal{B}$ of frames that have accumulated; streams that arrive from ESs do not accumulate a backlog. Figure 2.5 shows three streams, f_1 to f_3, and only f_3 arrives from a switch and hence will have a backlog of frames captured by the stream denoted with f_3' (we consider a $\mathcal{B}$ of 1 in the example). We also assume that the backlog f_3' will not arrive at the same time as the original stream f_3, and instead, it is delayed by a period. Again, this is a heuristic used for pruning, and the actual schedulability check is done with the NC analysis. So, the definition of the "transmission demand" does not impact correctness, but, as discussed, it will impact our algorithm's ability to search for solutions.

Since, in our case, the deadlines can be larger than the periods, we also need to consider, for each stream, bursts of frames coming from SWs and an additional frame for each stream coming from ESs (the ES periods are not synchronized with the SWs GCLs). Since we do not perform a worst-case analysis, we instead use a backlog parameter $\mathcal{B}$, capturing the possible number of delayed frames in a burst within a stream forwarded from another SW. Note that, as explained in the overview at the beginning of Section 2.5, $\mathcal{B}$ is a user-defined parameter that controls the "pruning level" of our timing constraint, i.e., how aggressively it eliminates candidates from the search space.

For example, in Figure 2.5, the streams $f_1 < 50,5,0,5 >$ and $f_2 < 60,6,0,6 >$ have been received from an ES and the stream $f_3 < 100,15,0,15 >$ has been received from a SW. For the stream f_3 forwarded from a previous switch, we consider that one instance of the stream (determined by the backlog parameter $\mathcal{B} = 1$), let us call it f_3', may have been delayed and received together with the current instance f_3. This would cause a delay in the reception of the streams in the current node. The reception curve in Figure 2.5 is the sum of curves for each stream separately in a hyperperiod of 30 μs.

We give the general definition of the transmission demand value $\mathcal{W}_D$ as the area under the curve for the accumulated data amount of received streams and backlogs of the streams arrived from switches in a hyperperiod. For calculating the transmission demand $\mathcal{W}_D$, we consider 2 terms that are A_1 and A_2. The term A_1 calculates the area under the curve for the accumulated data of all streams assigned to the queue q captured by $\mathcal{R}(q)$, in a hyperperiod. Any frames of all streams $\mathcal{R}(q)$ have arrived at the beginning of their period. The term A_2 calculates the area under the curve for the accumulated backlog data of the streams arrived from a switch captured by $\mathcal{X}(q)$. The backlog data of the streams $\mathcal{X}(q)$ are delayed for a period and controlled by $\mathcal{B}$, which captures the number of backlogs. The function $\mathcal{W}_D$ returns 16,650 $Bytes \times \mu s$ in our example, see also Figure 2.5 for the values of the terms A_1 and A_2.

$$
\begin{aligned}
&\forall p \in P, \forall q \in p.Q, \forall f \in \mathcal{R}(q), \forall f' \in \mathcal{X}(q): \\
&I = \frac{\mathcal{K}(p)}{f.T}, \quad I' = \frac{\mathcal{K}(p)}{f'.T} \\
&A_1 = \frac{I \times (I+1)}{2} \times f.T \times f.l, \\
&A_2 = \frac{I' \times (I'+1-2\times\mathcal{B})}{2} \times f'.T \times f'.l, \\
&\mathcal{W}_D = A_1 + A_2
\end{aligned}
\tag{2.10}
$$

Please note that the correctness of the constraints (Eq. (2.3), (2.4), (2.5)) follows from the implicit hardware constraints of 802.1Qbv (see the discussion in [CSCS16, SOCS18]) while other constraints (Eq. (2.6), (2.7)) are used to limit the placement of GCL windows and are not related to correctness, just to optimality. Since the transmission of frames is decoupled from the GCL windows, the schedule's correctness concerning the end-to-end latency of streams is always guaranteed due to the NC analysis, which is intertwined in the schedule step.

2.6 Simulated Annealing Window Optimization

The previously described CPWO delivers good results in a reasonable time for small problem sizes. However, for larger problem sizes, the method either becomes intractable, or the search space pruning has to be done very aggressively, leading to a degradation in the quality of the results. Therefore, we propose a heuristic algorithm that is aimed to be scalable for large problem sizes while still offering good quality solutions.

Several metaheuristic approaches have been proposed in the literature for intractable problems [BK^{+}05]. Based on the review of the related work, we have decided to develop a Simulated Annealing (SA)-based metaheuristic solution. Metaheuristics are designed to find good quality solutions while still being scalable for large problem sizes, but are not guaranteed to find an optimal (or any) solution.

Our Simulated Annealing Window Optimization (SAWO) algorithm is shown in Algorithm 1. The key feature of SA is that it avoids getting stuck in a local optimum by accepting worse intermediate solutions with a certain probability, which decreases throughout the search [KGJV83, BK^{+}05]. The likelihood of considering a worse solution (compared to the current solution) depends on the worsening of the objective function and a temperature parameter t [VW02]. In SA, the temperature starts from an initial temperature T_{start} (line 4) and is decreased in every iteration with a factor $0 < \alpha < 1$ (line 15).

SA starts from an initial solution Φ (line 2, see Section 2.6.1) which is evaluated using the objective function Ω^{SA} (line 3), see Section 2.6.2 for details. When there are no infeasible streams the objective functions for CPWO and SAWO are identical and thus comparable. If there are infeasible streams, we penalize this in the SAWO solution in order to guide the search (see Section 2.6.2).

SA iterates until a stopping criterion, like a timeout or iteration limit, is satisfied (lines 5–15). In every iteration, we generate a random "neighbor" of the current solution (line 6) and calculate the difference δ between its objective value Ω^{SA}_{new} and the objective value Ω^{SA} of the current solution (line 8). Section 2.6.3 presents how we generate a neighbor solution. If the new objective value is smaller, we accept the new solution as the current (and possibly best, see lines 12–14). However, we also sometimes accept a worse neighbor solution. This is the case, if a random value (between 0 and 1) is smaller than an "acceptance probability function" $e^{\frac{\delta}{t}}$ (see line 9). This acceptance probability function decreases with a larger δ, i.e., we are less likely to accept worse neighbors if they are further from the current solution, or a smaller temperature, i.e., the probability of accepting worse neighbors decreases during the search. We use a time limit as the stopping criterion.

Algorithm 1: Simulated Annealing Window Optimization (SAWO)

```
1  Function SAWO(𝓕,P)
2      Φ_best = Φ = InitialSolution(𝓕,P);
3      Ω^SA_best = Ω^SA = Objective(Φ);
4      t = T_start;
5      while stopping-criterion not True do
6          Φ_new = RandomNeighbor(Φ, p_mv);
7          Ω^SA_new = Objective(Φ_new);
8          δ = Ω^SA_new − Ω^SA;
9          if δ < 0 or random[0,1) < e^(δ/t) then
10             Φ = Φ_new;
11             Ω^SA = Ω^SA_new;
12             if Ω^SA_new < Ω^SA_best then
13                 Φ_best = Φ_new;
14                 Ω^SA_best = Ω^SA_new;
15         t = t ∗ α;
16     return Φ_best;
```

2.6.1 Initial Solution

The goal of the InitialSolution function, shown in Alg. 2, is to find a good starting point for SA. We start out by choosing a common period for all windows in the port, which helps speed up overlap and worst-case latency calculations. The choice of this period is important: A larger period means longer worst-case latencies but less bandwidth occupation, since the distance between two consecutive windows is longer (in the worst-case a stream arrives right at the moment when it can't fit into the current window anymore, requiring it to wait a full window period). We choose the minimum period that is larger than the combined length of all streams in the port from a set containing all stream periods, their greatest common divisor, and half of that value (line 3-6). Then we decide the length for each window (line 11). It has to be at least as long as the total sending time of all streams in that queue (line 9), and its size relative to the period of the window has to be at least as big as the stream sizes relative to their period (line 10). Finally, we align the windows in the different queues, so they do not overlap (line 12-13), since, in the worst-case, an overlapping part of a window has to be considered occupied.

Algorithm 2: SA Initial Solution

```
1  Function InitialSolution(𝓕, P)
2      foreach p ∈ P do
3          T^p = {f.T | q ∈ p.Q_ST, f ∈ 𝓕(q)};
4          foreach q ∈ p.Q_ST do
5              q.T = MinPossiblePeriod(T^p ∪ gcd(T^p) ∪ gcd(T^p)/2);
6          end
7          φ_cur = 0;
8          forall q ∈ p.Q_ST do
9              tl = sum({f.l | f ∈ 𝓕(q)});
10             pp = sum({f.l / f.T | f ∈ 𝓕(q)});
11             q.w = max(tl, pp * q.T) + max(F_l^q);
12             q.φ = φ_cur;
13             φ_cur = φ_cur + q.w;
14         end
15     end
16     return P;
```

2.6.2 SA Objective Function

The objective function Ω^{SA} used inside SA is shown in Equation 2.11. The difference between Ω^{SA} and Ω used by CP (Equation 2.1, Section 2.5.1) is that Ω minimizes the average bandwidth usage and then uses timing constraints and network calculus to check that a solution is schedulable.

Ω^{SA} has two components: The first component Ω^{bw} is measuring the bandwidth consumed by all windows across all ports on average, and is equivalent Ω. The second component Ω^{inf} is the number of streams that miss their deadline, also called infeasible streams. Thus, instead of using the schedulability as a constraint as in the CP formulation, we allow SA to visit unschedulable solutions in the hope of driving the search towards schedulable solutions. The amount of infeasible streams is determined by running the NC-based worst-case delay analysis of [ZPGF32] with the given set of windows. Since the average consumed bandwidth is at most 1 (equals to 100%), any missed deadlines will increase the objective value and thus drive the search to schedulable solutions that decrease Ω^{inf}, which dominates the objective function when a solution is not schedulable. The weights w_a and w_b in Equation 2.11 can be used to control the relative importance of the two components, e.g., when a system engineer prefers lower bandwidth at the expense of schedulability, w_a can be increased and w_b decreased. In our experiments, we have used $w_a = w_b = 1$, which we found is a good choice when searching for schedulable solutions.

$$\Omega^{SA} = w_a \times \Omega^{bw} + w_b \times \Omega^{inf}, \text{ where} \tag{2.11}$$
$$\Omega^{bw} = \frac{\sum \frac{q.w}{q.T}}{\mathcal{N}(P)}, \forall p \in P, \forall q \in p.Q$$
$$\Omega^{inf} = |\{f \mid f \in \mathcal{F} \wedge m_i^{dest(f)}.\phi + m_i^{dest(f)}.L - m_i^{src(f)}.\phi > f.D - \delta\}|$$

2.6.3 Neighbor Function

The purpose of the RandomNeighbor(Φ, p_{mv}) function in Algorithm 1 is to select a close neighbor of the current solution Φ. A neighbor is generated by performing a transformation (also called "moves") on the current solution. This transformation function is designed such that the SA search will have good coverage of the solution space. The solution space, in our case, includes all solutions that have one window per queue, with a minimum length and without overlap of windows in the same port. Our neighbor function uses two different moves to change the current solution:

- **MoveWindow**: Selects a random occupied queue. Changes the window offset to a random value within the range of all offsets where the window will not overlap with windows in other queues on the same port. This move occurs with a given probability of p_{mv}.
- **ChangeWindowSize**: Selects a random occupied queue. Changes the window size to a random value in the range between the minimum window size and the maximum size before an overlap with another window in the same port would occur. This move is applied with a probability of $1 - p_{mv}$.

We have used a value of $p_{mv} = 0.8$ in the experiments. This makes it more likely for a MoveWindow move to occur, which is usually more impactful, since the window sizes are already set to reasonable initial values by the initial solution, while the window alignment across ports is not very good yet.

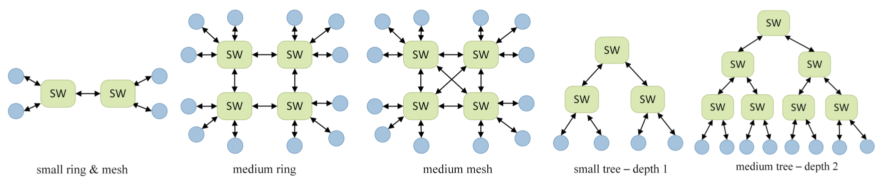


Figure 2.6: Network Topologies used in the Test Cases [ZPS17]-

2.7 Evaluation

In this section, we evaluate our optimization solutions CPWO and SAWO for *FWND* on synthetic and real-world test cases in Section 2.7.1 and Section 2.7.2, respectively.

2.7.1 CPWO Evaluation

Test Cases and Setup

We implemented our *CPWO* approach using the Java version of Google OR-Tools and the Java kernel of the RTC toolbox [Goo20, WT06b]. The tests were run on an i9 CPU (3.6 GHz) with 32 GB of memory. The timeout for is set to 10 to 90 minutes, depending on the size of the test case. The macrotick and $\mathcal{B}$ parameters are set to 1 μs and 1, respectively, in all the test cases.

We have generated 15 synthetic test cases that have different network topologies (three test cases for each topology in Figure 2.6) inspired by industrial and automotive application requirements. Similar to [ZPS17], the network topologies are small ring & mesh (SRM), medium ring (MR), medium mesh (MM), small tree-depth 1 (ST), and medium tree-depth 2 (MT). The message sizes of streams are randomly chosen between 64 bytes and 1518 bytes, while their periods are selected from the set $P = \{1{,}500, 2{,}500, 3{,}500, 5{,}000, 7{,}500, 10{,}000\}\mu$s. The physical link speed is set for 100 Mbps. The details of the synthetic test cases are in Table 2.4 where the second column shows the topology of the test cases, and the number of switches, end systems, and streams are shown in columns 3 to 6.

We have also used two realistic test cases: an automotive case from General Motors (GM) and an aerospace case, the Orion Crew Exploration Vehicle (CEV). The GM case consists of 27 streams varying in size between 100 and 1,500 bytes, with periods between 1 *ms* and 40 *ms* and deadlines smaller or equal to the respective periods. The CEV case is larger, consisting of 137 streams, with sizes ranging from 87 to 1,527

Table 2.4: Details of the Synthetic Test Cases.

No.	Network Topology	Total No. of SWs	Total No. of ESs	Total No. of Streams	Hyperperiod (μs)
1	SRM	2	3	9	15,000
2	SRM	3	3	11	70,000
3	SRM	3	4	15	70,000
4	MR	4	6	15	30,000
5	MR	4	8	21	210,000
6	MR	5	11	27	210,000
7	MM	4	5	13	15,000
8	MM	6	12	30	210,000
9	MM	7	13	35	210,000
10	ST	3	4	7	15,000
11	ST	3	6	12	15,000
12	ST	3	7	16	105,000
13	MT	7	8	18	105,000
14	MT	7	8	25	105,000
15	MT	7	12	32	210,000

bytes, periods between 4 *ms* and 375 *ms*, and deadlines smaller or equal to the respective periods. The physical link speed is set for 1000 Mbps. More information can be found in the corresponding columns in Table 2.6. Use cases use the same topologies as in [GZPS17] and [ZPGF32], and we consider that all streams are ST.

CPWO Evaluation on Synthetic Test Cases

We have evaluated our CPWO solution for *FWND* on synthetic test cases. The results are depicted in Table 2.5 where we show the objective function value (average bandwidth Ω from Equation 2.1) and the mean WCDs. For a quantitative comparison, we have also reported the results for the three other ST scheduling approaches: *0GCL*, *FGCL*, *WND*. *0GCL* and *FGCL* were implemented by us with a CP formulation using the constraints from [CSCS16] and [SOCS18], respectively. The *WND* method has been implemented with the heuristic presented in [RZCP20], but instead of using the WCD analysis from [ZPC18], we extend it to use the analysis from [ZPGF32] instead, in order not to unfairly disadvantage WND over our CPWO solution. Note that the respective mean worst-case end-to-end delays in the table are obtained over all the streams in a test case, from a single run of the algorithms, since the output of the algorithms is deterministic based on worst-case analyses, not based on simulations.

It is important to note that *0GCL* and *FGCL* are presented here as a means to evaluate CPWO; however, they are *not producing valid solutions* for our problem, which con-

Table 2.5: CPWO Evaluation Results on Synthetic Test Cases.

No.	Ω^1 for 0GCL	Ω^1 for FGCL	Ω^1 for WND	Ω^1 for CPWO	Mean worst-case e2e-delay for 0GCL (μs)	Mean worst-case e2e-delay for FGCL(μs)	Mean worst-case e2e-delay for WND (μs)	Mean worst-case e2e-delay for CPWO (μs)	Mean Runtime for 0GCL (ms)	Mean Runtime for CPWO (ms)
1	35	35	614	510	192	126	1838	1556	215	8/164
2	25	22	640	528	246	151	2461	1806	895	12/249
3	15	15	549	495	175	486	1964	1384	1518	22/203
4	13	13	330	285	160	776	2925	1832	525	16/338
5	14	NA[2]	295	285	131	NA[2]	2838	2347	5187	16/423
6	13	NA[2]	275	205	129	NA[2]	2953	1976	6291	17/721
7	12	12	238	204	125	764	2913	1561	1152	35/3235
8	13	NA[2]	238	202	114	NA[2]	2878	1725	7603	36/2075
9	12	NA[2]	217	191	122	NA[2]	3074	1927	9171	56/12084
10	8	8	329	265	136	2284	4397	4327	2611	165/325
11	10	10	381	302	159	984	3047	2057	2840	231/1552
12	11	NA[2]	516	321	187	NA[2]	2543	1326	4650	260/3393
13	10	10	401	302	101	561	2529	471	978	130/914
14	9	9	611	402	120	785	2254	628	1256	162/1077
15	9	NA[2]	544	413	114	NA[2]	2680	713	6116	163/1433

[1] Values are multiplied by 1000
[2] Ran out of memory

siders unscheduled end systems, see Table 2.1 for the requirements of each method. As expected, when end systems are scheduled and synchronized with the rest of the network as is considered in *0GCL* and *FGCL*, we obtain the best results in terms of bandwidth usage (Ω) and WCDs, noting that *0GCL* may further reduce the WCDs compared to *FGCL*.

The only other approach that has similar assumptions to our *CPWO* approach is *WND* from [RZCP20]. As we can see from Table 2.5, in comparison to *WND*, our CPWO solution can slightly reduce the bandwidth usage. The most important result is that CPWO significantly reduces the WCDs compared to *WND*, with an average of 104% and up to 437% for some test cases such as TC13. Hence, we are able to obtain schedulable solutions in more cases compared to the work in [RZCP20]. Also, when comparing the WCDs obtained by our *CPWO* approach with the case when the end systems are scheduled, i.e., *0GCL* and *FGCL*, we can see that the increase in WCDs is not dramatic. This means that for many classes of applications, which can tolerate a slight increase in latency, we can use our CPWO approach to provide solutions for more types of network implementations, including those that have unscheduled and/or unsynchronized end systems. In addition, due to the complexity of their CP model, it takes a long runtime to obtain solutions for *0GCL* and *FGCL*, and the CP-model for *FGCL* run out of memory for some of the test cases (the NA in the table). As shown in the last two columns of Table 2.5, where we present the runtimes of *0GCL* and *CPWO*, *CPWO* reduces the runtime significantly. The two numbers in the runtime column represent the runtime for the obtaining the last solution and the runtime for the whole CPWO run, respectively. The reason for reduced runtime with *CPWO* is that the CP model has to determine values for fewer variables compared to *0GCL*. *CPWO* introduces 3 variables (offset, period and length) for each window (queue) in the network, whereas *0GCL* introduces a variable for each frame of each stream. The number of variables in the *0GCL* model depends on the hyperperiod, the number of streams, and the stream periods, whereas the number of variables in the *CPWO* model depends on the number of switches and used queues.

Table 2.6: CPWO Results on Realistic Test Cases.

	ORION (CEV)	**GM**
ES	31	20
SW	15	20
Streams	137	27
Mean WCDs (μs)	10,376	1,981
Ω ($\times$1000)	435	84
Runtime (s)	891	17

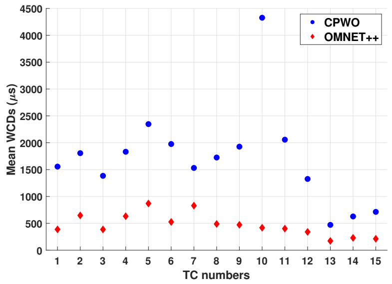
CPWO Evaluation on Realistic Test Cases

We have used two realistic test cases to investigate the scalability of CPWO and its ability to produce schedulable solutions for real-life applications. The results of the evaluation are presented in Table 2.6 where the mean WCDs, objective value Ω, and runtime for the two test cases are given. As we can see, CPWO has successfully scheduled all the streams in both test cases. Note that once all streams are schedulable, CPWO aims at minimizing the bandwidth. This means that CPWO may be able to achieve even smaller WCD values at the expense of bandwidth usage. In terms o runtime, the CEV test case takes longer since it has 864 variables, whereas GM has only 102 variables in the CP models.

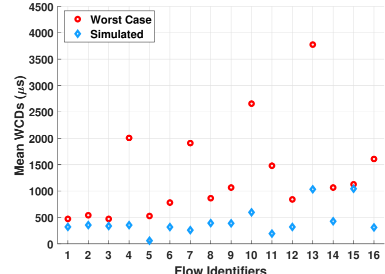
Note that we had to use a very aggressive pruning parameter $\mathcal{B} = 1$ for all the CPWO experiments. This was the only way to ensure that CPWO terminates and returns a solution. A more "relaxed" $\mathcal{B}$ will result in CPWO running for several days without returning a solution. Due to the aggressive pruning, the solution returned by CPWO is not guaranteed to find the optimal solution and will, in fact, as the next section will show, miss good quality solutions.

Validating the CPWO Solutions with OMNET++

We have used the OMNET++ simulator with the TSN NeSTiNg extension [FHC$^+$19] to validate the generated GCLs. Thus, we have synthesized the GCLs for all approaches on all synthetic test cases, and we have observed that the GCLs are correct, and the simulation behaves as expected. The mean WCDs of CPWO for the synthetic test cases and the worst-case latency observed during multiple OMNET++ simulations (with the windows from CPWO) are depicted in Figure 2.7a. As expected, the latency values reported by OMNET++ are smaller than the WCDs, as reported by the WCD Analysis from [ZPGF32]. This is because a simulation cannot easily uncover the worst-case behavior. However, the simulation indicates the average behavior, and small delays mean that even for unscheduled/unsynchronized end systems, we are able to obtain so-



(a) Mean WCDs vs. Simulated Delays for CPWO.



(b) Mean WCDs Comparison for the Streams of TC12,

Figure 2.7: WCD vs. Simulated Delays.

lutions that are not only schedulable (WCDs are smaller than the deadlines) but also have good average behavior, where most of the time the delays are reasonable, even smaller than the static schedules obtained by *0GCL* and *CPWO* for scheduled and synchronized ESs. The pessimism result of the WCD analysis is unavoidable in systems with un-synchronized and/or unscheduled end-systems; in practice, however, simulated delays are much smaller, as can be seen in Figure 2.7a and Figure 2.7b. We also show in Figure 2.7b the simulated delays and WCDs for all streams of TC12. All the streams are schedulable, and, as expected, the simulated delays are smaller than the WCDs, calculated with the worst-case delay analysis derived in the work from [ZPGF32].

CPWO Scalability Evaluation

We have investigated the scalability of CPWO on 6 larger test cases (TC1 to TC6), that have up to 120 devices (75 ESs and 45 SWs) and 500 streams. The results and the details of the test cases are presented in Table 2.7, where columns 2, 3, and 4 show the number of streams, end-systems, and switches, respectively. Columns 5, 6, and 7

Table 2.7: Scalability Evaluation of CPWO.

No.	Total No. of Streams	Total No. of ESs	Total No. of SWs	Mean WCDs μs	Largest deadline μs	Ω ($\times 1000$)
TC1	100	50	35	3,226	4,000	249
TC2	150	55	40	3,521	4,000	366
TC3	200	60	40	4,387	5,000	396
TC4	300	65	40	4,911	6,000	468
TC5	400	70	45	5,210	6,000	498
TC6	500	75	45	4,399	5,000	511

show the mean WCD of streams in μs, the largest deadline of all streams in μs, and the objective value Ω, related to bandwidth, see Equation 2.1. CPWO was able to generate schedulable solutions in all cases. Furthermore, CPWO is optimize the schedules for minimum bandwidth usage and has generated solutions that, besides being schedulable, have mean WCDs on average 14% smaller than the respective deadlines in all test cases.

2.7.2 SAWO Evaluation

Test Cases and Setup

For the comparison of SAWO and CPWO in Section 2.7.2 and the SAWO evaluation using realistic test cases in Section 2.7.2, we used the same test cases defined in Section 2.7.1. For the test cases in Section 2.7.2 we have taken the topology sizes proposed in [CSO16] as a reference. We implemented our SAWO solution in Python and configured it for all experiments with $a = b = 1, p_{mv} = 0.8$ on an i7-8565U CPU with 16GB memory and using Python 3. The solution communicates with the worst-case delay analysis of [ZPGF32] via socket, eliminating unnecessary delay of file I/O.

SAWO Comparison to CPWO

We compare the SA-based approach (SAWO) to the CP-based solution (CPWO) but also to the classical zero-jitter GCL (0GCL) [CSCS16, PLCS16], Frame-to-Window-based GCL (FGCL) [SOCS18], and Window-based GCL (WND) [RZCP20] solutions, in terms of the objective value (i.e., quality of the solution) and the mean worst-case end-to-end latency for the streams. We do not show the runtime figures, since the SA-based solution was always set to a runtime of 2 and 10 min. We note that the runtime for CPWO is small due to the aggressive pruning parameter, thus trading off the quality of the solution for algorithm runtime. For the experiments, we use the same synthetic test cases described in Section 2.7.1. The details of the synthetic test cases can be found in Table 2.4.

Table 2.8 presents the results for SAWO compared to the aforementioned solutions. For SAWO, the columns showing the objective value Ω and the mean worst-case e2e delay present two numbers obtained with 2 and 10 minute runtime, respectively.

We can see that SAWO can achieve significantly better results than CPWO in terms of the objective value Ω. While the runtime of 2 min is sufficient to get a good result, this result can be further improved by a longer runtime (see Section 2.7.2 for a more detailed analysis of the runtime impact). The objective function does not include the

Table 2.8: SAWO Evaluation Results on Synthetic Test Cases.

No.	Ω^1 for 0GCL	Ω^1 for FGCL	Ω^1 for WND	Ω^1 for CPWO	Ω^1 for SAWO	Mean worst-case e2e-delay for 0GCL (μs)	Mean worst-case e2e-delay for FGCL(μs)	Mean worst-case e2e-delay for WND (μs)	Mean worst-case e2e-delay for CPWO (μs)	Mean worst-case e2e-delay for SAWO (μs)
1	35	35	614	510	452/455	192	126	1838	1556	1412/1424
2	25	22	640	528	346/343	246	151	2461	1806	1648/1797
3	15	15	549	495	300/293	175	486	1964	1384	1684/1752
4	13	13	330	285	236/222	160	776	2925	1832	1608/1488
5	14	NA[2]	295	285	250/225	131	NA[2]	2838	2347	1380/1526
6	13	NA[2]	275	205	254/208	129	NA[2]	2953	1976	1250/1407
7	12	12	238	204	136/92	125	764	2913	1561	848/784
8	13	NA[2]	238	202	225/144	114	NA[2]	2878	1725	787/981
9	12	NA[2]	217	191	218/146	122	NA[2]	3074	1927	774/1138
10	8	8	329	265	85/84	136	2284	4397	4327	4787/4509
11	10	10	381	302	104/97	159	984	3047	2057	2472/2518
12	11	NA[2]	516	321	254/251	187	NA[2]	2543	1326	1367/1188
13	10	10	401	302	157/166	101	561	2529	471	1149/898
14	9	9	611	402	196/201	120	785	2254	628	1219/1057
15	9	NA[2]	544	413	210/206	114	NA[2]	2680	713	1045/1081

[1] Values are multiplied by 1000
[2] Ran out of memory

mean e2e-delay but the number of infeasible streams. That means that it is beneficial for the solutions to accept a higher mean e2e-delay for a lower objective value, e.g., by decreasing the size of a window. That can be seen, for example, in test case 2. However, sometimes there are also solutions that have both a lower objective value and e2e-delay, e.g., test case 4. This can happen through a window being moved to a better offset, which would decrease the e2e-delay without increasing the objective value. Please note that the objective function Ω is the same for both CPWO and SAWO since SAWO can schedule all streams and thus there is no penalty term for infeasible streams in the SAWO objective function.

As mentioned before in Section 2.7.1, *0GCL* and *FGCL* are included as a means to evaluate SAWO; however, they are *not producing valid solutions* for our problem, as they require scheduled end-systems. As expected, when end systems are scheduled and synchronized with the rest of the network, as is considered in *0GCL* and *FGCL*, we obtain the best results in terms of bandwidth usage (Ω) and WCDs.

SAWO Evaluation on Realistic Test Cases

Table 2.9: SAWO Results on Realistic Test Cases.

	ORION (CEV)	GM
ES	31	20
SW	15	20
Streams	137	27
Mean WCDs (μs)	341	992
Ω ($\times 1000$)	374	15
Runtime (s)	600	600

As with CPWO, we have used two realistic test cases from [GZPS17] and [ZPGF32],

an automotive case from General Motors (GM) and an aerospace case, the Orion Crew Exploration Vehicle (CEV), where we consider that all streams are critical and scheduled. For the details of the test cases, please see the description in Section 2.7.1. We show the scalability of SAWO and its ability to produce schedulable solutions for real-life applications. The results of the evaluation are presented in Table 2.9 where the mean WCDs, objective value Ω, and runtime for the two test cases are given. As a comparison to CPWO, we refer the reader to the results presented in Table 2.6. As we can see, SAWO has successfully scheduled all the streams in both test cases and produces better results than CPWO in terms of mean WCD and quality of the solution (objective value Ω). The runtime for SAWO was set to 10 minutes for the two realistic test cases.

SAWO Evaluation on Large Synthetic Test Cases

As previously described, CPWO delivers good results in a reasonable time for small problem sizes, but does not scale well for large inputs unless the search space pruning is done very aggressively, which leads to low-quality solution. Therefore, we show that our SAWO heuristic algorithm scales well with the network and problem size while still offering good quality solutions.

To evaluate the impact of the heuristic runtime and the test case size on the resulting solution quality, we have created three test batches as described in Table 2.10. Each batch consists of 50 test cases with a mesh topology (see Figure 2.6) with sizes medium, large, and huge as described in [CSO16]. For each test case, we generated streams with random routes, priorities, and sizes under the constraint that no link utilization may exceed 50% until an average link utilization threshold of 15% was reached. For the medium test cases, there were between 31 and 67 streams with an average of around 46 streams per test case. For the large batch, the 50 test cases had between 127 and 178 streams averaging 147 streams. The huge test batches averaged 416 streams per test case with a minimum of 364 and a maximum of 475. Each stream has a random size between 64 and 1500 Bytes and a random period from the set $\{1,2,5,10\}$ *ms*, as defined for the use cases in [KZH15]. The stream deadline is set to ten times the stream's period.

Table 2.10: Parameters of SAWO Test Batches.

	Topology	Number of testcases	SW	ES	Avg. number of streams
medium	mesh	50	4	16	46
large	mesh	50	8	48	148
huge	mesh	50	16	96	416

We ran each test batch with a 2-minute and a 10-minute timeout and measured the best objective function value Ω^{SA} obtained within the timeout. Figure 2.8 shows the results as box plots for the medium and large test batches (y-axis), with all objective values multiplied by 1000 for clarity (x-axis). We set the upper and lower whisker bounds to depict outliers above $1.5 \times IQR$ of the 3rd quartile and under $1.5 \times IQR$ of the 1st quartile. Additionally, we show all data points within the figure. The median for the medium size test-cases with 2 minutes and 10 minutes timeout was 172.5 and 140, respectively. The median for the large test cases with 2 minutes and 10 minutes timeout was 180 and 179, respectively.

We can see that the test cases are consistently completely schedulable (objective value below 1000) with good solution quality. Furthermore, we can see that the heuristic quickly can find good quality solutions. A longer runtime has a positive impact on the solution quality, but this impact depends on the size of the test case. The time needed to achieve significant improvement increases with the size of the test case, as the amount of possible moves increases in parallel with the worst-case analysis taking more time per iteration.

Figure 2.9 shows the result for the huge test batches as a box plot with the same whisker boundaries and outlier setting as before but with a logarithmic x-axis showing the objective value. The median for the huge test cases were 180.5 and 179.5 for the 2 and 10 minute timeout, respectively. With a 2 minute timeout, 13 out of 50 test cases had at least one unschedulable stream (objective value over 1000) and overall low solution quality. With a 10 min timeout, the solution quality improved, and only 8 out of 50 test cases had at least one unschedulable stream. From the total of 20810 streams in the 50 test cases, a total of 35 streams were unschedulable with a 2 minute timeout, while a total of 21 were unschedulable with the 10 minute timeout.

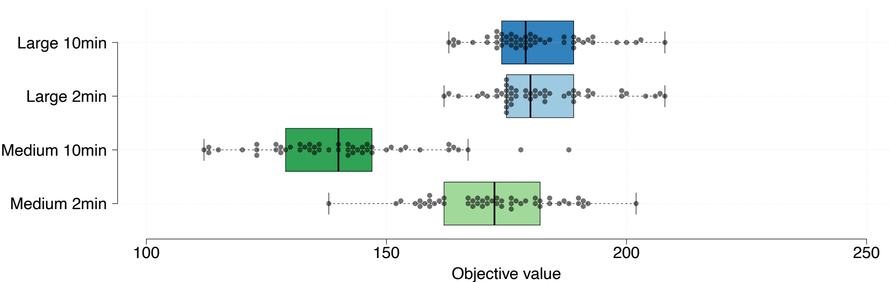


Figure 2.8: Objective Value Boxplots for Medium and Large SAWO Test Batches.

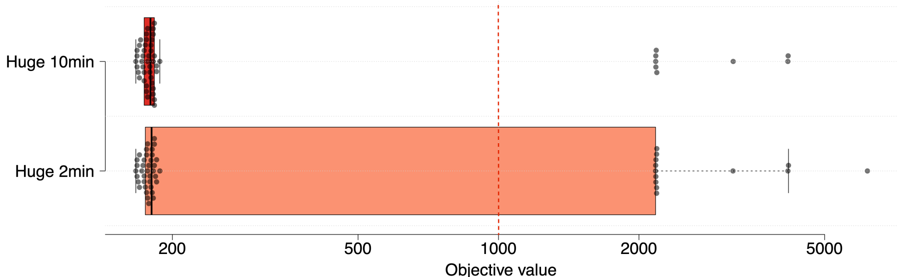


Figure 2.9: Objective Value Boxplots for Huge SAWO Test Batches.

2.8 Conclusions

We have addressed the problem of guaranteeing real-time communication behavior in heterogeneous TSN networks, introducing a more flexible heuristic schedule synthesis approach (FWND) which decouples the frame transmission from the scheduled time-aware shaper (TAS) windows. Using this approach, we have proposed two solutions to solve the problem, one based on a Constraint-Programming formulation within a Tabu Search metaheuristic (CPWO) and one based on a Simulated Annealing metaheuristic (SAWO). The CPWO solution uses a novel proxy function that can be parametrized to trade off run-time performance for search-space pruning in the CP-model. We have shown that for large use cases, CPWO has to either aggressively prune the search space, leading to low-quality solutions, or is intractable. Therefore, we have introduced SAWO, which scales better for large test cases while still offering good quality solutions. We evaluated our approaches using synthetic and real-world test cases, comparing them with existing mechanisms, and validated the generated schedules using OMNET++.

CHAPTER 3

Paper B: Dependability-Aware Routing and Scheduling for Time-Sensitive Networking

Time-Sensitive Networking (TSN) extends IEEE 802.1 Ethernet for safety-critical and real-time applications in several areas, e.g., automotive, aerospace or industrial automation. However, many of these systems also have stringent security requirements, and security attacks may impair safety. Given a TSN-based distributed architecture, a set of applications with tasks and messages, as well as a set of security and redundancy requirements, we are interested to synthesize a system configuration such that the real-time, safety and security requirements are upheld. We use the Timed Efficient Stream Loss-Tolerant Authentication (TESLA) low-resource multicast authentication protocol to guarantee the security requirements, and redundant disjunct message routes to tolerate link failures. We consider that tasks are dispatched using a static cyclic schedule table and that the messages use the time-sensitive traffic class in TSN, which relies on schedule tables (called Gate Control Lists, GCLs) in the network switches. A configuration consists of the schedule tables for tasks, as well as the disjoint routes and GCLs for messages. We propose a Constraint Programming-based formulation which can be used to find an optimal solution with respect to our cost function. Additionally,

we propose a Simulated Annealing based metaheuristic, which can find good solution for large test cases. We evaluate both approaches on several test cases.

3.1 Introduction

Many modern safety-critical real-time systems are implemented on distributed architectures. They integrate various software functions with different security and safety requirements over the same deterministic communication network. For example, the network in a modern vehicle has to integrate high-bandwidth video and LIDAR data for Advanced Driver Assistance Systems (ADAS) functions with the highly critical but low bandwidth traffic of e.g., the powertrain functions, but also with the best-effort messages of the low-criticality diagnostic services [GP20]. Figure 3.1 presents an example of an ADAS network architecture with redundant routes.

Time-Sensitive Networking (TSN) [Ins16c], which is becoming the standard for communication in several application areas, e.g., automotive to industrial control, is comprised of a set of amendments and additions to the IEEE 802.1 standard, equipping Ethernet with the capabilities to handle real-time mixed-criticality traffic with high bandwidth. A TSN network consists of several end-systems that run mixed-criticality applications interconnected via network switches and physical links. Available traffic types are Time-Triggered (TT) traffic for real-time applications, Audio-Video Bridging (AVB) for communication that requires less stringent bounded latency guarantees, and Best-Effort (BE) traffic for non-critical traffic [GZRP18].

We assume that safety-critical applications are scheduled using static cyclic scheduling and use the TT traffic type with a given *Redundancy Level* (RL) for communication. We consider that the task-level redundancy is addressed using solutions such as replication [IPEP05], and we instead focus on the safety and security of the communication in TSN. The real-time safety requirements of critical traffic in TSN networks are enforced through offline-computed schedule tables, called Gate Control Lists (GCLs), that specify the sending and forwarding times of all critical frames in the network. Scheduling time-sensitive traffic in TSN is non-trivial (and fundamentally different from e.g., TTEthernet) because TSN does not schedule communication at the level of individual frames as is the case in TTEthernet. Instead, the static schedule tables (GCLs) govern the behavior of entire traffic classes (queues), which may lead to non-deterministic frame transmissions [CSCS16].

Since link and connector failures in TSN could result in fatal consequences, the network topology uses redundancy, e.g., derived with methods such as [GZPS17]. In TSN, IEEE 802.1CB Frame Replication and Elimination for Reliability (FRER) enables the transmission of duplicate frames over different (disjoint) routes, implementing merging

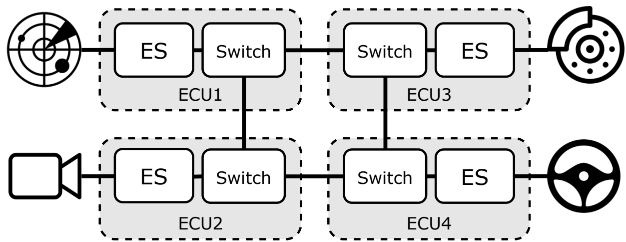


Figure 3.1: Automotive TSN-based CPS with Redundant Routes.

of frames and discarding of duplicates.

Nowadays, modern Cyber-Physical Systems (CPSs) are becoming increasingly more interconnected with the outside world opening new attack vectors [PBA17, SNA$^+$13] that may also compromise safety. Therefore, the security aspects should be equally important to the safety aspects. Timed Efficient Stream Loss-Tolerant Authentication (TESLA) [PCST01] has been investigated as a low-resource authentication protocol for several networks, such as FlexRay and TTEthernet [ZQLY19] networks. However, adding security mechanisms such as TESLA after the scheduling stage is often not possible without breaking real-time constraints, e.g., on end-to-end latency, and degrading the performance of the system [ZQLY19]. Thus we consider TESLA and the overhead and constraints it imposes as part of our configuration synthesis problem formulation.

3.1.1 Related Work

Scheduling for TSN networks is a well-researched problem. It has been solved for a variety of different traffic type combinations (TT, AVB, BE) and device capabilities using methods such as Integer Linear Programming (ILP), Satisfiability Modulo Theories (SMT) or various metaheuristics such as tabu search [CSCS16, SOCS18, DN16, GZRP18, ZSEP21a, HAD$^+$21, SALC21, VHT21, VBHT22].

Routing has also been extensively researched [WH00, GHKS98]. In [SBCH13] the authors presented an ILP solution to solve the routing problem for safety-critical AFDX networks. In [TSPS15] the authors used a tabu search metaheuristic to solve the combined routing and scheduling problem for TT traffic in TTEthernet. In [PD12] the authors provide a simple set of constraints to solve a general multicast routing problem using constraint programming, which [GZPS17] builds on that to solve a combined topology and route synthesis problem. In [OY20] the authors use a load-balancing heuristic to distribute the bandwidth usage over the network and achieve lower latency for critical traffic.

Multiple authors have also looked at the combined routing and scheduling problem. The authors in [LPS16] and [NDR18b] showed that they are able to significantly reduce the latency by solving the combined problem with an ILP formulation. In [PTO19] the authors presented a heuristic for a more complex application model that allows multicast streams. They were able to solve problems that were infeasible to solve using ILP or separate routing and scheduling.

Recently authors have started to present security- and redundancy-aware problem formulations. The authors in [ZQLY19] provided a security-aware scheduling formulation for TTEthernet using TESLA for authentication. In [MAS$^+$19] the authors solve the combined routing and scheduling problem and considered authentication using block ciphers. The authors in [HWW$^+$21] and [AHM20], on the other hand, present a routing and scheduling formulation that is redundancy-aware but has no security considerations.

To the best of our knowledge, our work is the first to provide a formulation that is both security and redundancy-aware.

3.1.2 Contributions

In this paper, we address TSN-based distributed safety-critical systems and solve the problem of configuration synthesis such that both safety and security aspects are considered. Determining an optimized configuration means deciding on the schedule tables for tasks as well as the disjoint routes and GCLs for messages. Our contributions are the following:

1. We apply TESLA to TSN networks considering both the timing constraints imposed by TSN and the security constraints imposed by TESLA.
2. We formulate an optimization problem to determine: (i) the redundant routing of all messages; (ii) the schedule of all messages, encapsulated into Ethernet frames, represented by the GCLs in the network devices, and (iii) the schedule of all related tasks on end-systems.
3. We extend our Constraint Programming (CP) formulation from [RPC20] and propose a new Simulated Annealing (SA)-based metaheuristic to tackle large-scale networks that cannot be solved with CP
4. We evaluate the impact of adding the security from TESLA on the schedulability of applications, and we evaluate the solution quality and scalability of the Constraint Programming (CP) and Simulated Annealing (SA) optimization approaches

We introduce the fundamental concepts of TSN in Section 3.2 and of TESLA in Section 3.3. In Section 3.4 we present the model of our system, consisting of the network

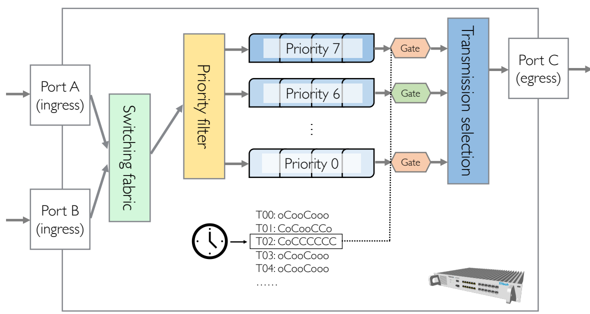


Figure 3.2: Simplified TSN Switch Representation.

architecture and applications running on this architecture. Additionally we present a threat model and how it is addressed by TESLA with a security model. In Section 3.5 we formulate the problem we are solving using the established models and present an example. In Section 3.6 and Section 3.7 we present the two different optimization approaches, CP and SA. Then, we evaluate these approaches using several test cases in Section 3.8. Section 3.9 concludes the paper.

3.2 Time-Sensitive Networking

Time-Sensitive Networking [Ins16c] has arisen out of the need to have more stringent real-time communication capabilities within standard Ethernet networks. Other technologies that offer real-time guarantees for distributed systems are TTEthernet (SAE AS6802 [Iss11, SBHP11]), PROFINET, and EtherCAT [Pry08]. TSN comprises a set of (sub-)standards and amendments for the IEEE 802.1Q standard, introducing several new mechanisms for Ethernet bridges, extensions to the IEEE 802.3 media access control (MAC) layer, as well as other standards and protocols (e.g., 802.1ASrev). A survey on the research results and the standardization efforts for TSN can be found in [SHM$^+$21].

The fundamental mechanisms that enable deterministic temporal behavior over Ethernet are, on the one hand, the clock synchronization protocol defined in IEEE 802.1ASrev [Ins17a], which provides a common clock reference with bounded deviation for all nodes in the network, and on the other hand, the timed-gate functionality (IEEE 802.1Qbv [Ins16b]) enhancing the transmission selection on egress ports. The timed-gate functionality (IEEE 802.1Qbv [Ins16b]) enables the predictable transmission of communication streams according to the predefined times encoded in schedules called Gate-Control Lists (GCL). A stream in TSN definition is a communication carrying

a certain payload size from a talker (sender) to one or multiple listeners (receivers), which may or may not have timing requirements. In the case of critical streams, the communication has a defined period and a maximum allowed end-to-end latency.

Other amendments within TSN (c.f. [Ins16c]) provide additional mechanisms that can be used either in conjunction with 802.1Qbv or stand-alone. IEEE 802.1CB [Ins17b] enables stream identification, based on e.g., the destination MAC and VLAN-tag fields in the frame, as well as frame replication and elimination for redundant transmission. IEEE 802.1Qbu [Ins16a] enables preemption modes for mixed-criticality traffic, allowing express frames to preempt lower-priority traffic. IEEE 802.1Qci [Ins17c] defines frame metering, filtering, and time-based policing mechanisms on a per-stream basis using the stream identification function defined in 802.1CB.

We detail the Time-Aware Shaper (TAS) mechanism defined in IEEE 802.1Qbv [Ins16b] via the simplified representation of a TSN switch in Figure 3.2. The figure presents a scenario in which communication received on one of two available ingress ports (A and B) will be routed to an egress port C. The switching fabric will determine, based on internal routing tables and stream properties, to which egress port a frame belonging to the respective stream will be routed (in our logical representation, there is only one egress port). Each port will have a priority filter that determines which of the available 8 traffic classes (priorities) of that port the frame will be enqueued in. This selection will be made based on either the PCP field of the 802.1Q VLAN-tag of frames or the *stream gate instance table* of 802.1Qci, which can be used to circumvent traffic class assignment of the PCP code. As opposed to regular 802.1Q bridges, where the transmission selection sends enqueued frames according to their respective priority, in 802.1Qbv bridges, there is a Time-Aware Shaper (TAS), also called timed-gate, associated with each traffic class queue and positioned before the transmission selection algorithm. A timed-gate can be either in an *open (o)* or *closed (C)* state. When the gate is open, traffic from the respected queue is allowed to be transmitted, while a closed gate will not allow the respective queue to be selected for transmission, even if the queue is not empty. The state of the queues is encoded in a local schedule called Gate-Control List (GCL). Each entry defines a time value and a state (o or C) for each of the 8 queues. Hence whenever the local clock reaches the specified time, the timed-gates will be changed to the respective open or closed state. If multiple nonempty queues are open simultaneously, the transmission selection selects the queue with the highest priority for transmission.

The Time-Aware Shaper functionality of 802.1Qbv, together with the synchronization protocol defined in 802.1ASrev, enables a global communication schedule that orchestrates the transmission of frames across the network such that real-time constraints (usually end-to-end latencies) are fulfilled. The GCL schedule synthesis problem has been addressed in [CSCS16, SOCS18, PLCS16, DN16] for ensuring deterministic communication behavior for critical streams.

Craciunas et al. [CSCS16] define correctness conditions for generating GCL schedules, resulting in a strictly deterministic transmission of frames with 0 jitter. Apart from technological constraints, e.g., only one frame transmitted on a link at a time, the deterministic behavior over TSN is enforced in [CSCS16] through isolation constraints. Since the TAS determines the temporal behavior of entire traffic classes and not of individual frames, the queue state always has to be deterministic and hence, [CSCS16] enforces a strict isolation of critical streams by not allowing two critical streams to be enqueued in the same queue at the same time. This condition is called *frame/stream isolation* in [CSCS16]. In [SOCS18], critical streams are allowed to overlap to some degree (determined by a given jitter requirement) in the same queue in the time domain, thus relaxing the strict isolation.

Both approaches enforce that gate states of different scheduled queues are mutually exclusive, i.e., only one gate is open at any time, thus preventing the transmission selection from sending frames based on their assigned traffic class's priority. By circumventing the priority mechanism through the TAS, it is ensured that no additional delay is produced through streams of higher priorities, thus enforcing a highly deterministic temporal behavior.

3.3 Timed Efficient Stream Loss-Tolerant Authentication

TESLA provides a resource-efficient way to perform asymmetric authentication in a multicast setting [PCST01]. It is described in detail in [PCST01] and [PSC$^+$05].

We are considering systems where one end-system wants to send a multicast signal to multiple receiver end-systems, e.g., periodic sensor data. A message authentication code (MAC), which is appended to each signal, can guarantee authenticity, i.e., that the sender is whom he claims to be, and integrity, i.e., that the message has not been altered. The MAC is generated and authenticated by a secret key that all end-systems share (i.e., symmetric authentication). The downside of this approach is that if any of the receiving end-systems is compromised, the attacker would be able to masquerade as the sender by knowing the secret key. In a multicast setting, an asymmetric approach, in which the receivers do not have to trust each other, is preferable.

The traditional asymmetric authentication approach is to use asymmetric cryptography with digital signatures (i.e., private and public keys); however, as stated in [PCTS02], the method is computationally intensive and not well suited for systems with limited resources and strict timing constraints.

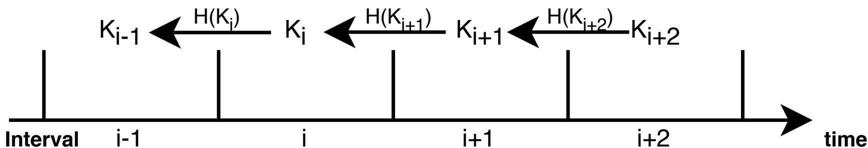


Figure 3.3: TESLA Key Chain (adapted from [PCTS02]).

However, TESLA uses an approach where the source of asymmetry is a time-delayed key disclosure [PCST01]. Although this can be implemented with much less overhead, it requires time synchronization between the network nodes. For TSN, the time synchronization is given through the 802.1ASrev protocol.

Figure 3.3 visualizes the TESLA protocol. As described in [PCTS02], when using TESLA, time is divided into fixed intervals of length P_{int}. At startup a one-way chain of self authenticating keys K_i is generated using a hash function H, where $K_i = H(K_{i+1})$. Each key is assigned to one interval. The protocol is bootstrapped by creating this chain and securely distributing K_0 to all receivers [ZQLY19].

Normally in TESLA, as described in [ZQLY19], when a sender sends a message m in the *i-th* interval, it appends to that message: i, a keyed-MAC using the key of that interval K_i, and a previously used key K_{i-d}. Thus, a key remains secret for d intervals. When a receiver receives a message m in the interval i it can not yet authenticate it and must wait until a message arrives in the interval $i+d$. This message discloses K_i, which can be used to decrypt the MAC of m and thus authenticate it. To ensure that K_i itself is valid, we can use any previously validated key. For e.g., we can check that $H(K_i) = K_{i-1}$, $H(H(K_i)) = K_{i-2}$ etc. This makes TESLA also robust to packet loss since any lost keys can be reconstructed from a later key, and any key can always be checked against K_0.

Due to the deterministic nature of our schedule, we can make some modifications to the basic TESLA protocol without sacrificing security. The first modification is adopted from [ZQLY19]. Since bandwidth is scarce, we do not release the key K_{i-d} with every message/stream. Instead, it will be released once in its own stream with an appropriate redundancy level. The second modification concerns the TESLA parameter d, the key disclosure delay. This parameter is useful in a non-deterministic setting since the arrival time of a stream is uncertain. A high value for d means that the corresponding key is released later, making it more likely that a stream can be authenticated, at the cost of increased latency [PSC+05]. However, in our case, we know the exact time a stream will be sent and arrive. Thus, we assume that a stream's keyed-MAC will be generated using the key from the interval it *arrives at the last receiver*. We will always release the key K_i in the interval $i+1$, minimizing the key disclosure delay and thus the latency before a stream can be authenticated.

3.4 System Models

This section presents the architecture and application models, as well as the threat, security, and fault models. Our application model is similar to the one used in related work [ZQLY19], but we have extended it to consider TSN networks and the optimization of redundant routing in conjunction with scheduling.

3.4.1 Architecture Model

We model our TSN network as a directed graph consisting of a set of nodes $\mathcal{N}$ and a set of edges $\mathcal{L}$. The nodes of the graph are either end-systems (ESs) or switches (SWs): $\mathcal{N} = \mathcal{ES} \cup \mathcal{SW}$. The edges $\mathcal{L}$ of the graph represent the network links.

We assume that all of the nodes in the network are TSN-capable, specifically that they support the standards 802.1ASrev [Ins17a] and 802.1Qbv [Ins16b]. Thus, we assume the whole network, including the end-systems, to be time-synchronized with a known bounded precision δ. All nodes use the time-aware shaper mechanism from 802.1Qbv to control the traffic flow.

Each end-system $e_i \in \mathcal{ES}$ features a real-time operating system with a periodic table-driven task scheduler. Hash computations, which will be necessary for TESLA operations on that end-system, take $e_i.H$ µs.

A network link between nodes $n_a \in \mathcal{N}$ and $n_b \in \mathcal{N}$ is defined as $l_{a,b} \in \mathcal{L}$. Since in Ethernet-compliant networks all links are bi-directional and full-duplex, we have that for each $l_{a,b} \in \mathcal{L}$ there is also $l_{b,a} \in \mathcal{L}$. A link $l_{a,b} \in \mathcal{L}$ is defined by a link speed $l_{a,b}.s$.

Figure 3.4a shows a small example architecture with four end-systems, two switches, and full-duplex links.

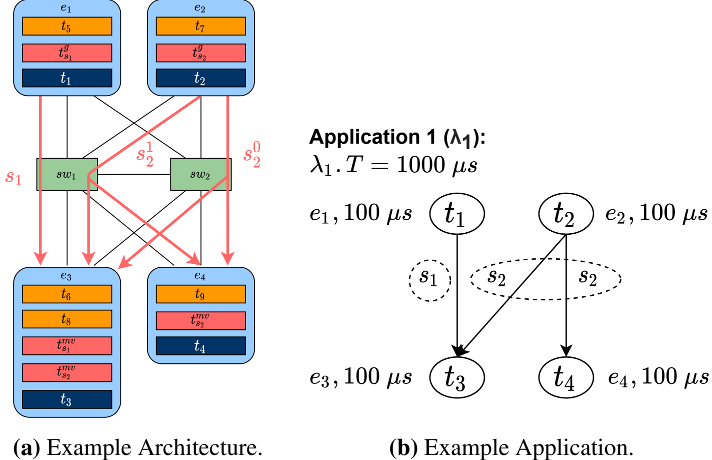
3.4.2 Application Model

An application $\lambda_l \in \Lambda$ is modeled as a directed, acyclic graph consisting of a set of nodes representing tasks Γ_l and a set of edges $\mathcal{E}_l$ representing a data dependency between tasks.

A task is executed on a certain end-system $t_m.e$. The worst-case execution time (WCET) of a task is defined by $t_m.w$ µs. A task needs all its incoming streams (incoming edges in the application graph) to arrive before it can be executed. It produces outgoing

Table 3.1: System Model Notations.

Description	Notation	Unit
Header overhead	OH	Byte
Maximum transmission unit	MTU	Byte
TESLA key size	KS	Byte
TESLA MAC size	MAC	Byte
Hyperperiod	H	µs
TSN Network Graph	$(\mathcal{N}, \mathcal{L})$	
- Nodes	$\mathcal{N} = \mathcal{ES} \cup \mathcal{SW}$	
- End-system	$e_i \in \mathcal{ES}$	
- Hash computation time	$e_i.H$	µs
- Switch	$sw_j \in \mathcal{SW}$	
- Links	$\mathcal{L} \subseteq \mathcal{N} \times \mathcal{N}$	
- Network link	$l_{a,b}$	
- Link speed	$l_{a,b}.s$	µs
Application	$\lambda_l \in \Lambda$	
- Tuple	$(\Gamma_l, \mathcal{E}_l)$	
- Period	$\lambda_l.T$	µs
- Communication Depth	$\lambda_l.C$	
- Tasks	$t_m \in \mathcal{T}$	
- Execution end-system	$t_m.e$	
- Worst-case execution time	$t_m.w$	µs
- Period	$t_m.T$	µs
- Streams	$s_n \in \mathcal{S}$	
- Source task	$s_n.t_s$	
- Destination tasks	$s_n.T_d$	
- Size	$s_n.b$	Byte
- Period	$s_n.T$	µs
- Redundancy Level	$s_n.rl$	
- Security Level	$s_n.sl$	
- MAC generation task	$t_{s_n}^{g}$	
- MAC verification task	$t_{s_n}^{mv}$	
Security Application	$\lambda_l^s \in \Lambda^{sec}$	
- Key release task	t_m^r	
- Key verification task	t_m^v	
- Key source end-system	$t_m^v.src$	
- Key stream	$s_n^k \in \mathcal{S}_k$	



(a) Example Architecture. **(b)** Example Application.

Figure 3.4: Example Architecture and Application Models.

streams at the end of its execution time. Communication dependencies between tasks that run on the same end-system are usually done via, e.g., shared memory or message queues, where the overhead of reading/writing data is negligible and included in the WCET definition of the respective tasks. Dependencies between tasks on separate end-systems constitute communication requirements and are modeled by streams. A stream in the TSN context is a communication requirement between a sender and one (unicast) or multiple (multicast) receivers. An example application can be seen in Figure 3.4b. An application is periodic with a period $\lambda_l.T$, which is inherited by all its tasks and streams.

A stream s_n originates at a source task $s_n.t_s$ and travels to set of destination tasks $s_n.T_d$ (since we consider multicast streams). The stream size $s_n.b$ is assumed to be smaller than the MTU (maximum transmission unit) defined for the network. Each stream has a redundancy level $s_n.rl$, which determines the amount of required disjunct redundant routes for the stream to take. For each of these routes we model a sub-stream: $s_n^i \in S_{s_n}, 0 \leq i < s_n.rl$ Hereby S_{s_n} is a set containing all sub-streams of s_n. This notation is useful for differentiating the different routes a stream takes through the network and making sure those routes do not overlap. A stream also has a binary security level $s_n.sl$ which determines if it is authenticated using TESLA ($sl = 1$) or not ($sl = 0$).

We define the hyperperiod H as the least-common multiple of all application periods: $H = lcm(\{\lambda_l.T | \lambda_l \in \Lambda\})$ We define the set $\mathcal{T}$ to contain all tasks and the set $\mathcal{S}$ to contain all streams (including redundant copies).

3.4.3 Fault Model

Reliability models discussed in [GZPS17] (e.g., Siemens SN 29500) indicate that the most common type of permanent hardware failures is due to link failures (especially physical connectors) and that ESs and SWs are less likely to fail. These models are complementary to the mean time to failure (MTTF) targets established, for example, in the automotive domain within the SIL levels of the ISO 26262 certification standard [GZPS17]. As mentioned, we assume that we know the required redundancy level to protect against permanent link failures. Our disjoint routing can guarantee the transmission of a stream of RL n despite any $n-1$ link failures. For example, for the routing of s_2 with RL 2 in Figure 3.4a, any 1-link failure would still result in a successful transmission.

3.4.4 Threat Model

We use a similar threat model to [ZQLY19] and assume that an attacker is capable of gaining access to some end-systems of our system, e.g., through an external gateway or physical access.

We consider that the attackers have the following abilities:

- They know about the network schedule and the content of the streams on the network;
- They can replay streams sent by other ES;
- They can attempt to masquerade as other ES by faking the source address of streams they send;
- They have access to all keys released and received by the ES they control;

3.4.5 Security Model

We use TESLA to address the threats identified in the previous section, which means that additional security-related models are required. These additional applications, tasks and streams can be automatically generated from a given architecture and application model.

First off, we need to generate, send, and verify a key in each interval for each set of communicating end-systems. We generate a key authentication application $\lambda_s \in$

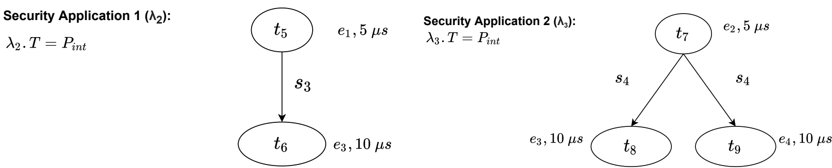


Figure 3.5: Example Security Model for the Applications in Figure 3.4.

Λ^{sec} for each sender end-system, which is modeled similarly to a normal application as a directed acyclic graph. The period $\lambda_s.T$ is equal to P_{int} (see Section 3.3) and again inherited by tasks and streams. Each of these applications consists of one key release task t_m^r scheduled on the sending end-system e_i. Additionally, it consists of key verification tasks t_j^v on each end-system e_j that receives a stream from e_i. The release task sends a multicast key-stream s_n^k to each of those verification tasks. The redundancy level of a key-stream $s_n^k.rl$ is set to the maximum redundancy level of all streams emitted by e_i. The size of a key stream $s_n^k.b$ is equal to the key size KS specified by the TESLA implementation. The security model for our example from Figure 3.4 can be seen in Figure 3.5.

For a key verification task $t_m^v.src$ is the end-system e_i whose key this task is verifying. Its execution time is equal to the length of one hash execution on its execution end-system: $t_m^v.w = (t_m^v.e).H$. The execution time of a key release task is very short since the key it releases has already been generated during bootstrapping. We model it to be half the time of a hash execution: $t_m^r.w = \frac{(t_m^r.e).H}{2}$

Secondly, we need to append MACs to all non-key-streams with $s_n.sl = 1$. Thus, their length increases by the MAC length MAC specified by the TESLA implementation. For each stream s_n, a MAC generation task $t_{s_n}^g$ is added to the sender and a MAC validation task $t_{s_n}^{mv}$ to each receiver. Those tasks take the time of one MAC computation on the processing element to execute.

We define the set $\mathcal{T}_{kr}^n$ to contain all key release tasks and $\mathcal{T}_{kv}^n$ to contain all key verification tasks for a given node n. Furthermore, let S_k contain all key streams.

Figure 3.4a shows key release and verification tasks in orange and MAC generation and validation tasks in red.

Figure 3.5 shows the security applications for our example.

3.5 Problem Formulation

Given a set of applications running on TSN-capable end-systems that are interconnected in a TSN network as described in the architecture, application and security models in Section 3.4, we want to determine a system configuration consisting of:

- an interval duration P_{int} for TESLA operations,
- the routing of streams,
- the task schedule,
- the network schedule as 802.1Qbv Gate-Control Lists (GCLs),

such that:

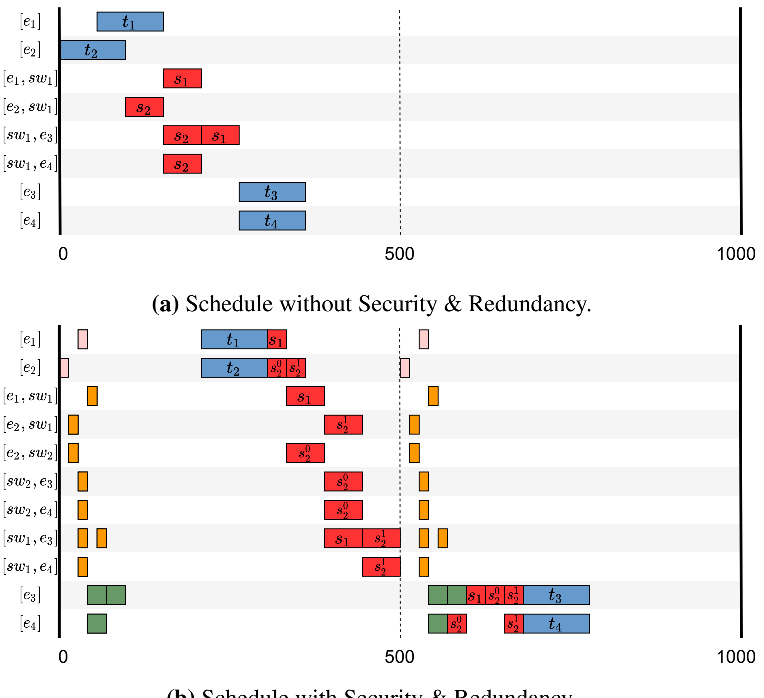
- all deadline requirements of all applications are satisfied.
- the redundancy requirements of all streams and the security conditions of TESLA are fulfilled.
- the overall latency of applications is minimized.

3.5.1 Motivational Example

We illustrate the problem using the architecture and application from Figure 3.4. We have one application Figure 3.4b with 4 tasks, 2 streams, and a period and deadline of 1000 µs. The tasks are mapped to the end-systems as indicated in the figure. Stream s_2 will be multicast. The size of both streams is 50 B. For TESLA's security requirements, i.e. $s_1.sl = s_2.sl = 1$, we generate two additional security applications (Figure 3.5).

We have a TSN network with a link speed of 10 Mbit/s and zero propagation delay. Our TESLA implementation uses keys that are 16 B and MACs that are 16 B. A hash computation takes 10 µs on every ES.

A solution that does not consider security and redundancy requirements is shown in Figure 3.6a. With the TSN stream isolation constraint outlined in Section 3.2 taken into account, the GCLs are equivalent to frame schedules. We depict in Figure 3.6a the GCLs as a Gantt chart, where the red rectangles show the transmission of streams s_1 and s_2 on network links, and the blue rectangles show the tasks' execution on the respective end-systems. To guarantee deterministic message transmission in TSN, we



(a) Schedule without Security & Redundancy.

(b) Schedule with Security & Redundancy.

Figure 3.6: Example Solution Schedules for the Models in Figure 3.4.

have to isolate the frames in the time (or space) domain, leading to the delay of s_1 and thus t_3. We refer the reader to [CSCS16] for an in-depth discussion on the non-determinism problem and isolation solution in TSN.

In this paper, we are interested in solutions such as the one in Figure 3.6b, which considers both the redundancy and security requirements. The black dashed line in the figure separates the TESLA key release intervals, where P_{int} was determined to be 500 µs. Streams carrying keys are orange, key generation tasks pink, key verification tasks green, and the MAC generation/validation operations on ESs are shown in red. The routing of the non-key streams can be seen in Figure 3.4a. Note how the two redundant copies of s_2, s_2^0 and s_2^1 use non-overlapping paths.

The delay incurred by the time-delayed release of keys is particularly important: tasks t_3 and t_4 can only be executed after the keys authenticating s_1 and s_2 have arrived in the second interval, and after key verification and MAC validation tasks have been run.

Scheduling problems like the one addressed in this paper are NP-hard as they can be reduced to the Bin-Packing problem [FDR18] and may be intractable for large input

X	**s1**	**s2_0**	**s2_1**
ES1	ES1	nil	nil
ES2	nil	ES2	ES2
ES3	SW1	SW1	SW2
ES4	nil	SW1	SW2
SW1	ES1	ES2	nil
SW2	nil	nil	ES2

Table 3.2: Matrix X for Example from Section 3.5.1.

sizes. In the following sections, we will propose a Constraint Programming (CP) formulation to solve the problem optimally for small test cases and a heuristic to solve the problem for large test cases.

3.6 Constraint Programming Formulation

Constraint Programming (CP) is a technique to solve combinatorial problems defined using sets of constraints on decision variables. For large scheduling problems, it becomes intractable to use CP due to the exponential increase in the size of the solution space [RVBW06]. In order to achieve reasonable runtime performance, we split the problem into 3 sub-problems which we solve sequentially: (i) finding a route for all streams, (ii) finding P_{int}, and (iii) finding the network and task schedule.

3.6.1 Optimizing Redundant Routing

The first step of solving the proposed problem is to find a set of (partially) disjoint routes for each stream, depending on the stream's redundancy level. The constraints in this section are inspired by [GZPS17] and [PD12].

We model the stream routes with an integer matrix X, where the columns represent streams (including their redundant copies) and rows represent nodes of the network. An entry at the position of a stream s_n and a node n in this matrix referring to a node m, represents a link from m to n on the route of stream s_n. Alternatively, the entry could be *nil*, in which case n is not part of the route.

Using the matrix X, we can construct the route for each stream bottom-up as a tree by starting at the receiver nodes. See Table 3.2 for the matrix of our example.

To determine the route for each stream $s_n \in \mathcal{S}$, for each node $n \in \mathcal{N}$ we have the following optimization variables:

- $x(s_n,n)$ represents an entry of our matrix X. The domain of $x(s_n,n)$ is defined as: $D(x(s_n,n)) = \{m \in \mathcal{N} | l_{m,n} \in \mathcal{L}\} \cup \{n\} \cup \{nil\}$. We refer to $x(s_n,n)$ as the successor of n on the path to the stream sender node.

- $y(s_n,n)$ represents the length of the path from n to $s_n.t_s.e$, i.e. the length of the path from node n to the sender node of the stream. $D(y(s_n,n)) = \{i | 0 <= i <= |\mathcal{SW}| + 1\}$

Furthermore, we define a few helper variables and functions. First off, we define $\mathcal{S}^d$ as the set of all distinct streams, i.e., excluding the redundant copies of streams with redundancy level (RL) greater than one. Additionally, we define $\mathcal{S}_{s_d}$ as the set of all redundant copies (including the stream itself) of s_d. Then we define the following helper function:

$$xsum(s_d,n,m) = \sum_{s'_d \in \mathcal{S}_{s_d}} (x(s'_d,n) == m) \tag{3.1}$$

This function allows us, for any given $s_d \in S^d$, to determine the number of redundant copies (including s_d itself) that use the link from m to n (nil is counted as zero).

Then we have the following constraint optimization problem:

$$Minimize: \sum_{s_n \in \mathcal{S}} cost(s_n) \tag{RC1}$$

where

$$cost(s_n) = \sum_{n \in \mathcal{N} \setminus \{s_n.t_s.e\}} (x(s_n,n)! = nil) \tag{RC2}$$

s.t.

$$x(s_n,n) \neq nil \Rightarrow y(s_n,n) = y(f,x(s_n,n)) + 1, \quad \forall s_n \in \mathcal{S},\ n \in \mathcal{N} \setminus \{s_n.t_s.e\} \tag{R1}$$

$$x(s_n,m) = nil \Leftrightarrow x(s_n,n) \neq m, \quad \forall s_n \in \mathcal{S},\ n,m \in \mathcal{N} \tag{R2}$$

$$x(s_n,n) \neq nil, \quad \forall s_n \in \mathcal{S},\ n \in \{t_r.e | t_r \in s_n.T_d\} \tag{R3.1}$$

$$x(s_n,s_n.t_s.e) = s_n.t_s.e, \quad \forall s_n \in \mathcal{S} \tag{R3.2}$$

$$x(s_n,n) = nil, \quad \forall s_n \in \mathcal{S},\ n \in \mathcal{ES} \setminus \{t_r.e | t_r \in s_n.T_d\} \tag{R3.3}$$

$$y(s_n,s_n.t_s.e) = 0, \quad \forall s_n \in \mathcal{S} \tag{R4}$$

$$\sum_{s_d \in \mathcal{S}^d} \Big((xsum(s_d,n,m) > 0\Big) \times \frac{s_d.b}{s_d.T}) \leq [m,n].s, \quad n,m \in \mathcal{N} \tag{R5}$$

$$x(s_n,n) \neq x(s'_n,n), \quad \forall s_n \in \mathcal{S},\ s'_n \in \mathcal{S}_{s_n} \setminus s_n,\ n \in \mathcal{N} \setminus \{s_n.t_s.e\}, \tag{R6}$$

Please note that == and != are boolean expressions that evaluate to 1 if true and to 0 otherwise.

The cost function we are minimizing ((RC1),(RC2)) measures the length of the route of each stream.[1]

The constraint (R1) prevents cycles in the route, as defined in [PD12]. The constraint (R2) disallows "loose ends", i.e., a node that has a successor/predecessor must have a predecessor/successor itself. Please note that we refer to the successor on the path from receiver to sender, i.e., the predecessor on the route. The constraint (R3.1) states that all receivers of a stream have to have a successor. Constraints (R3.2), (R3.3), and (R4) impose that the sender of the stream has itself as the successor, no other end-system has a successor, and the path length is 0 at the sender node, respectively. The constraint (R5) restricts the bandwidth usage of each link to be under 100%. If multiple copies of the same stream use the same link, only one of them is counted as consuming bandwidth, since we assume that streams are intelligently split and merged using IEEE 802.1CB. The constraint (R6) forbids the routes of redundant copies of a stream to overlap at any point.

[1]For some use cases, fully disjoint routes are not necessary. Refer to Section 3.10 for an updated formulation for this case

3.6.2 Optimizing P_{int}

To set up the TESLA protocol, we need to choose the parameter P_{int}. P_{int} is the duration of one key disclosure interval. It has a big influence on the latency of secure streams and thus on the feasibility/quality of the schedule.

When choosing P_{int} there is a trade-off between overhead and latency. A small P_{int} reduces the latency of secure streams but necessitates more key generation/verification tasks and key streams. Thus, we want to determine the maximum value of P_{int} for which the latency is still within all deadline bounds. To this end, we formulate constraints inspired by [ZQLY19] for which we then determine the optimal solution. This value is used as a constant in the subsequent optimization of the schedule.

We introduce a new notation: For each application $\lambda_l \in \Lambda$ we define $\lambda_l.C$ to be the communication depth, i.e., the length of the longest path in the application graph where only edges with associated secure streams are counted (ES-internal dependencies and non-secure streams are ignored). This gives us a measure of the longest chain of secure communications within the application, which we can use to estimate the amount of necessary TESLA intervals. Then we have:

$$Maximize : P_{int} \tag{P0}$$

s.t.

$$\forall \lambda_l \in \Lambda, \quad P_{int} \cdot (\lambda_l.C + 1) \leq \lambda_l.T \tag{P1}$$

$$H \ mod \ P_{int} = 0 \tag{P2}$$

$$P_{int} \ mod \ gcd(\{\lambda_l.T | \lambda_l \in \Lambda\}) = 0 \quad or$$

$$P_{int} * n = gcd(\{\lambda_l.T | \lambda_l \in \Lambda\}), \quad n \in \mathbb{N} \tag{P3}$$

The constraint (P1) guarantees that P_{int} is small enough to accommodate the authentication of all secure streams for all applications. The communication depth $\lambda_l.C$ of an application gives a lower bound of how many TESLA intervals are necessary to accommodate all these streams within the period of the application since there have to be $n+1$ intervals to accommodate the authentication of n secure streams.

The purpose of the constraints (P2) and (P3) is to align the TESLA intervals with the schedule. The (P2) makes P_{int} a divisor of the hyperperiod, while constraint (P3) makes P_{int} either a multiple or a divisor of the greatest common divisor of all application periods.

3.6.3 Optimizing Scheduling

In this step, we want to find a schedule for all tasks and streams which minimizes the overall latency of streams while fulfilling all constraints imposed by deadlines, TESLA, and TSN. The routes for each stream and P_{int} are given by the previous scheduling steps and assumed constant here.

We define the following integer optimization variables:

- o_l^s: offset of stream s on link or node l
- c_l^s: transmission duration of stream s on link or node l
- a_l^s: end-time of stream s on link or node l
- φ^s: index of the earliest interval where stream s can be authenticated on any receiver
- o_n^t: offset of task t (on node $t.e$)
- a_n^t: end-time of task t (on node $t.e$)

As an example, let us assume a hyperperiod of 1000us and a stream s with a period of 500us. $o_l^s = 100$, $c_l^s = 50$, $a_l^s = 150$ would imply that the stream s is scheduled on link l in the following time intervals: (100, 150) and (600, 650).

We also define several helper variables. Let $\mathcal{E}^s$ be the set containing all receiver end-systems of stream s:

$$\mathcal{E}^s = \{t.e \mid t \in s.T_d\}$$

Let $\mathcal{R}^s$ be the set containing all links on the route of stream s as well as sender and receiver nodes:

$$\mathcal{R}^s = \{s.t_s.e\} \cup \mathcal{E}^s \cup \{l_{a,b} \mid x(s,b) = a,\ l_{a,b} \in \mathcal{L}\} \tag{3.2}$$

Using these helper functions we define the following constraint-optimization problem for the task and network scheduling step:

$$Minimize : \sum_{\lambda_l \in \Lambda} cost(\lambda_l) \tag{CS1}$$

where

$$cost(\lambda_l) = max(\{a^t \mid t \in \Gamma_l\}) - min(\{o^t \mid t \in \Gamma_l\}) \tag{CS2}$$

s.t.

$$cost(\lambda_l) \leq \lambda_l.T \quad \forall \lambda_l \in \Lambda \tag{S1}$$

$$o_l^s = c_l^s = a_l^s = 0, \quad \forall s \in \mathcal{S},\ l_{a,b} \in \mathcal{L},\ l_{a,b} \notin \mathcal{R}^s \tag{S2.1}$$

$$o_n^s = c_n^s = a_n^s = 0, \quad \forall s \in \mathcal{S},\ n \in \mathcal{N},\ n \notin \mathcal{R}^s \tag{S2.2}$$

$$o_l^s + c_l^s = a_l^s, \quad \forall s \in \mathcal{S},\ l_{a,b} \in \mathcal{L},\ l_{a,b} \in \mathcal{R}^s \tag{S3.1}$$

$$o_n^s + c_n^s = a_n^s, \quad \forall s \in \mathcal{S},\ n \in \mathcal{N},\ n \in \mathcal{R}^s \tag{S3.2}$$

$$c_l^s = \left\lceil \frac{s.b}{l.s} \right\rceil, \quad \forall s \in \mathcal{S},\ l_{a,b} \in \mathcal{L},\ l_{a,b} \in \mathcal{R}^s \tag{S4.1}$$

$$c_n^s = n.H, \quad \forall s \in \mathcal{S},\ n \in \mathcal{N} \cap \mathcal{R}^s, s.secure == 1 \tag{S4.2}$$

The cost function we are minimizing here ((CS1),(CS2)) is the sum of the end-to-end latencies of all applications, i.e. the distance between the start time of the earliest task and the end time of the latest task for each application. The constraint (S1) sets the deadline for the completion of an application to its period. The constraints (S2.1) and (S2.2) set all optimization variables to zero for every stream, for all nodes and links not part of its route. For all other links and nodes constraints, (S3.1) and (S3.2) set the end-time to be the sum of offset a length. For each link on the route of a stream constraint (S4.1) sets the length to be the byte-size of the stream divided by the link-speed. In constraint (S4.2) the length of secure streams on end-systems is set to the length of one hash-computation on that end-system, approximating the duration of MAC generation/verification.

$$\varphi^s > \left\lfloor \frac{a_n^s}{P_{int}} \right\rfloor, \quad \forall s \in \mathcal{S},\ l_{a,b} \in \mathcal{L} \cap \mathcal{R}^s,\ b \in \mathcal{E}^s,\ s.secure == 1 \tag{S5}$$

$$o_n^s \geq a^{t_{key}} + \varphi^s * P_{int} \quad \forall s \in \mathcal{S},\ n \in \mathcal{E}^s, s.secure = 1 \quad \forall t_{key} \in T_{kv}^n \tag{S6}$$

$$a^s_{l_{a,b}} \leq o^s_{l_{b,c}} \tag{S7.1}$$
$$\forall s \in \mathcal{S},\ l_{b,c} \in \mathcal{L} \cap \mathcal{R}^s$$
$$a = x(s,b)$$

$$a^s_a \leq o^s_{l_{a,b}} \tag{S7.2}$$
$$\forall s \in \mathcal{S},\ s.secure == 1,$$
$$l_{a,b} \in \{l_{a,b} \mid l_{a,b} \in \mathcal{L} \cap \mathcal{R}^s,\ a = s.t_s.e\}$$

$$a^s_{l_{a,b}} \leq o^s_b \tag{S7.3}$$
$$\forall s \in \mathcal{S},\ s.secure == 1,$$
$$l_{a,b} \in \{l_{a,b} \mid l_{a,b} \in \mathcal{L} \cap \mathcal{R}^s,\ b \in \mathcal{E}^s\}$$

In constraint (S5) the earliest authentication interval for a stream φ^s is bound to be after the latest stream transmission interval. In constraint (S6) the start time of the stream on any receiver end-system is then bound to be greater or equal to the start time of that interval plus the end-time of the necessary preceding key verification task. The constraints (S7.1), (S7.2) and (S7.3) make sure that every stream is scheduled consecutively along its route. Thus, constraint (S7.1) enforces the precedence among two links, (S7.2) among the MAC generation on the sender and the first link and (S7.3) among the last link and the following MAC verification.

$$(\alpha \times s_1.T + a^{s_1}_l \leq \beta \times s_2.T + o^{s_2}_l) \quad \vee \tag{S8}$$
$$(\beta \times s_2.T + a^{s_2}_l <= \alpha \times s_1.T + o^{s_1}_l)$$
$$\forall s_1, s_2 \in \mathcal{S}, s_1 \neq s_2, \forall l \in \mathcal{R}^s_1 \cap \mathcal{R}^s_2,$$
$$\forall \alpha \in \{0, ..., lcm(s_1.T, s_2.T)/s_1.T\},$$
$$\forall \beta \in \{0, ..., lcm(s_1.T, s_2.T)/s_2.T\}$$

$$(\alpha \times s_2.T + o^{s_2}_{l_{b,c}} <= \beta \times s_1.T + o^{s_1}_{l_{a_1,b}}) \quad \vee \tag{S9}$$
$$(\beta \times s_1.T + o^{s_1}_{l_{b,c}} <= \alpha \times s_2.T + o^{s_2}_{l_{a_2,b}})$$
$$\forall s_1, s_2 \in \mathcal{S}, s_1 \neq s_2, \forall l \in \mathcal{R}^s_1 \cap \mathcal{R}^s_2,$$
$$a_1 = x(s_1,b),\ a_2 = x(s_2,b),$$
$$\forall \alpha \in \{0, ..., lcm(s_1.T, s_2.T)/s_1.T\},$$
$$\forall \beta \in \{0, ..., lcm(s_1.T, s_2.T)/s_2.T\}$$

The constraint (S8) prevents any streams from overlapping on any nodes or links. Furthermore, constraint (S9) guarantees that for each link connected to an output port of

a switch, the frames arriving on all input ports of that switch that want to use this output port cannot overlap in the time domain. This is the frame isolation necessary for determinism in our TSN configuration, which is further explained in [CSCS16].

$$o^t + t.w = a^t, \qquad \forall t \in \mathcal{T} \tag{T1}$$

$$a^t \leq o^s_{t.e} \tag{T2.1}$$
$$\forall t \in \mathcal{T},\ s \in \mathcal{S},\ s.t_s = t,\ s.secure == 1$$

$$a^t \leq o^s_{l_{a,b}} \tag{T2.2}$$
$$\forall t \in \mathcal{T},\ s \in \mathcal{S},\ s.t_s = t,\ s.secure == 0$$
$$\forall l_{a,b} \in \mathcal{L} \cap \mathcal{R}^s,\ a == t.e$$

$$a^s_{t.e} \leq o^t \tag{T3.1}$$
$$\forall t \in \mathcal{T},\ s \in \mathcal{S},\ t \in s.T_d,\ s.secure == 1$$

$$a^s_{l_{a,b}} \leq o^t \tag{T3.2}$$
$$\forall t \in \mathcal{T},\ s \in \mathcal{S},\ t \in s.T_s,\ s.secure == 0$$
$$\forall l_{a,b} \in \mathcal{L} \cap \mathcal{R}^s,\ b \in \mathcal{E}^s$$

$$(\alpha \times t_1.T + a^{t_1} \leq \beta \times t_2.T + o^{t_2}) \quad \vee \tag{T4}$$
$$(\beta \times t_2.T + a^{t_2} \leq \alpha \times t_1.T + o^{t_1})$$
$$\forall t_1, t_2 \in \mathcal{T},\ t_1 \neq t_2,$$
$$\forall \alpha \in \{0, ..., lcm(t_1.T, t_2.T)/t_1.T\},$$
$$\forall \beta \in \{0, ..., lcm(t_1.T, t_2.T)/t_2.T\}$$

$$(\alpha \times t.T + a^t \leq \beta \times s.T + o^s_{t.e}) \quad \vee \tag{T5}$$
$$(\beta \times s.T + a^s_{t.e} \leq \alpha \times t.T + o^t)$$
$$\forall t \in \mathcal{T},\ s \in \mathcal{S}, s.secure == 1, t.e \in \mathcal{R}^s$$
$$\forall \alpha \in \{0, ..., lcm(t.T, s.T)/t.T\},$$
$$\forall \beta \in \{0, ..., lcm(t.T, s.T)/s.T\}$$

The constraint (T1) sets the end-time of a task to be the sum of offset and length. The constraints (T2.1) and (T2.2) model the dependency between a task and all its outgoing streams: such streams may only start after the task has finished. Similarly, constraints (T3.1) and (T3.2) model the dependency between a task and its incoming streams: such a task may only start after all incoming streams have arrived. Finally, constraint (T4) prevents any two tasks from overlapping, while constraint (T5) prevents a task from overlapping with a MAC generation/verification operation.

3.7 Metaheuristic Formulation

As mentioned in Section 3.6, the scheduling problem addressed in this paper is NP-hard. As a consequence, a pure CP formulation solved using a CP solver is not tractable for large problem sizes. Hence, in this section, we propose a metaheuristic-based strategy, which aims to find good solutions (without the guarantee of optimality) in a reasonable time, even for large test cases.

An overview of our strategy is presented in Algorithm 3. We use a Simulated Annealing (SA) metaheuristic [KGJV83] to find solutions $\Phi = (\mathcal{R}, \Sigma)$, consisting of a set of routes $\mathcal{R}$ and a schedule Σ. As an input, we provide our architecture model $(\mathcal{N}, \mathcal{L})$ and the application model Λ. SA randomly explorers the solution space in each iteration by generating "'neighbors" of the current solution using design transformations (or "moves"). We consider both routing and scheduling-related moves, and the choice is controlled by a p_{rmv} parameter that gives the probability of a routing move. To measure the quality of a solution, we use a cost function with two parameters, a and b, which are factors for punishing overlap of redundant streams and missed deadlines for applications, respectively. While we always accept better solutions, the central idea of Simulated Annealing is to also accept worse solutions with a certain probability in order not to get stuck in local optima [BK$^+$05].

Algorithm 3 shows the main loop of the heuristic. We start out with an initial solution, a cost value, and a positive temperature. (line 2-4). Then, we repeat the steps described below until a stopping criterion like a time- or iteration-limit is met. We create a slight permutation of the current solution Φ by using the *RandomNeighbour* function (line 6). We calculate the cost of the new solution (line 7) and a delta of the new and old cost (line 8). Now, if the delta is smaller than 0, i.e., if Φ_{new} is a better solution than Φ, we choose Φ_{new} as the current solution (line 10-12). Alternatively, the new solution is also accepted if a randomly chosen value between 0 and 1 is smaller than the value of the acceptance probability function $e^{-\frac{\delta}{t}}$. This acceptance probability will decrease with the temperature over time and is also influenced by δ, which gives a measure of how much worse the new solution is. Finally, since we will occasionally accept worse solutions, we keep track of the best cost achieved overall and adjust it if necessary (line 12-14).

3.7.1 Precedence Graph

We introduce a helper data structure in the form of a precedence graph. A precedence graph is a collection of special DAGs, one for each application. These DAGs are expanded versions of the DAGs from the application model. Here, streams are modeled as nodes instead of edges, and each redundant copy of a stream has its own node. See

Algorithm 3: Simulated Annealing Metaheuristic

1 **Function** *heuristic*$(\mathcal{N}, \mathcal{L}, \Lambda, T_{start}, \alpha, k, p_{rmv}, a, b, w)$
2 $\quad \Phi_{best} = \Phi =$ InitialSolution($\mathcal{N}, \mathcal{L}, \Lambda, k$);
3 $\quad c_{best} =$ c = Cost(Φ, a, b);
4 $\quad$ t = T_{start};
5 $\quad$ **while** *stopping-criterion not True* **do**
6 $\quad\quad \Phi_{new} =$ RandomNeighbour(Φ, p_{rmv});
7 $\quad\quad c_{new} =$ Cost(Φ_{new}, a, b);
8 $\quad\quad \delta = c_{new} - c$;
9 $\quad\quad$ **if** $\delta < 0$ ***or*** *random[0,1)* $< e^{-\frac{\delta}{t}}$ **then**
10 $\quad\quad\quad \Phi = \Phi_{new}$;
11 $\quad\quad\quad c = c_{new}$;
12 $\quad\quad\quad$ **if** $c_{new} < c_{best}$ **then**
13 $\quad\quad\quad\quad \Phi_{best} = \Phi_{new}$;
14 $\quad\quad\quad\quad c_{best} = c_{new}$
15 $\quad\quad$ t = $t * \alpha$;
16 $\quad$ **end**
17 $\quad$ **return** Φ_{best};

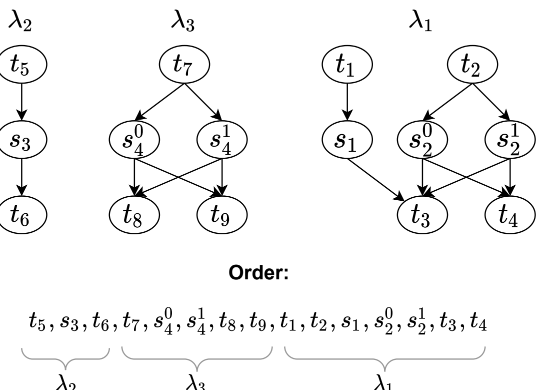


Figure 3.7: Example Precedence Graph with Associated Order.

Figure 3.7 for an example. This data structure helps to model all the dependencies between tasks and streams in the scheduling algorithm. Additionally, we will use the set of all topological orders of this graph as our solution space for the scheduling step. An order can be seen as a scheduling priority assignment that respects all precedence constraints.

3.7.2 Initial Solution

In the beginning, we create an initial solution Φ from the given architecture and application model. A solution is a tuple $(\mathcal{R}, \Sigma)$ consisting of a set of routes $\mathcal{R}$ and a schedule Σ. Algorithm 4 details the function to find the initial solution.

To find an initial set of routes, we iterate through all streams and all pairs of sender and receiver ES (lines 2-3). For each such pair, we calculate and store k shortest paths for the given topology (line 5). For each redundant copy of a stream beyond the first, we calculate the shortest path in a weighted graph, where we weight all links used by previous copies with w instead of 1 (line 7). For the initial solution, we choose the shortest path for each pair (line 8). Note that our k-shortest-path algorithm only generates paths without repeated nodes that do not traverse any end-system.

To find an initial schedule, we have to create the precedence graph P (line 11) and decide an order O of this graph.

For the initial solution, we construct an order on the level of applications, i.e., we avoid interleaving nodes of different applications. We prioritize key applications (lines 12-14) before other (normal) applications (lines 15-17). This order is consequently used to create a schedule (line 19). See Figure 3.7 for an example order.

3.7.3 Neighbourhood Function

The neighbourhood function $RandomNeighbour(\Lambda, p_{rmv})$ is detailed in Algorithm 5. It is used during Simulated Annealing to create a slight permutation of a given solution/candidate Φ. It contains two fundamental moves: Changing the routing $\Lambda.\mathcal{R}$ or changing the schedule $\Lambda.\Sigma$. Which move is taken is decided randomly (line 3). The parameter p_{rmv} influences how likely it is that the routing move is taken, e.g., $p_{rmv} = 0.5$ would result in a probability of 50%.

A routing move consists of choosing a random stream s out of the set of all streams (line 4), choosing a random receiver e_r out of all receivers of that stream (line 5) and then assigning a random path out of the set of k-shortest-paths calculated during the creation of the initial solution (line 6).

A scheduling move consists of choosing two random normal (non-key) applications d_1 and d_2 (lines 8, 9), switching their order O in the precedence graph P (line 10) and recalculating the schedule (line 11). Whenever a new schedule is calculated, we also optimize its latency (line 12). This is further explained in Section 3.7.6.

Algorithm 4: InitialSolution

```
1  Function InitialSolution(𝒩, ℒ, Λ, k, w)
       // routing
2      foreach s ∈ 𝒮 do
3          foreach e_r ∈ {t_r.e | t_r ∈ s.T_d} do
4              if IsFirstCopyOfStream(s) then
5                  K_s^{e_r} = ShortestPaths(s.t_s.e, e_r, k, 𝒩, ℒ);
6              else
7                  K_s^{e_r} = ShortestPathsWeighted(s.t_s.e, e_r, k, 𝒩, ℒ, w);
8              Φ.R^s = ShortestPath(K_s^{e_r}) ;
9          end
10     end
       // schedule
11     P = CreatePrecedenceGraph(Λ);
12     foreach λ_s ∈ Λ^sec do
13         O = O ∪ TopologicalOrder(λ_s, P);
14     end
15     foreach λ_n ∈ Λ \ Λ^sec do
16         O = O ∪ TopologicalOrder(λ_n, P);
17     end
18     Φ.K = K; Φ.P = P; Φ.O = O;
19     Φ.Σ = Schedule(O, Φ.ℛ);
20     return Φ;
```

Algorithm 5: RandomNeighbour

```
1  Function RandomNeighbour(Φ, p_rmv)
2      p = random[0, 1];
3      if p < p_rmv then
4          s = RandomStream(Φ);
5          e_r = RandomReceiver(s);
6          Φ.ℛ^s = RandomPath(Φ.K_s^{e_r});
7      else
8          d_1 = RandomNormalApplication(Φ);
9          d_2 = RandomNormalApplication(Φ);
10         Φ.O = SwitchSchedulingOrder(d_1, d_2, Φ.O);
11         Φ.Σ = Schedule(Φ.O, Φ.ℛ);
12         Φ.Σ = OptimizeLatency(Φ.Σ, Φ.P);
13     end
14     return Φ;
```

3.7.4 Cost Function

The cost function is used in the simulated annealing metaheuristic to evaluate the quality of a solution. A lower cost means a better solution. Algorithm 6 shows how our cost function is calculated. It consists of two components: a routing cost c_{route} and a schedule cost c_{sched}. The routing cost is the sum of the number of overlaps of redundant stream (one for each stream for each link) which is punished with a factor a and the total accrued length of all routes. The schedule cost is the sum of the number of infeasible applications, which is punished with a factor b, and the total sum of all application latencies (distance between start-time of first task and end-time of the last task). The factors a and b should be sufficiently high such that solutions with less overlap and infeasible applications are preferred.

Algorithm 6: Cost

```
1 Function Cost(Φ, a, b)
2     c_route = a * Overlaps(Φ.R) + Length(Φ.R);
3     c_sched = b * Infeasible(Φ.Σ) + Latency(Φ.Σ);
4     return c_route + c_sched;
```

3.7.5 ASAP List Scheduling

To calculate a schedule for a given precedence graph with associated order and routing, we use an ASAP list-scheduling heuristic [Sin07b], which schedules each node of the precedence graph in the given order.

The algorithm, presented in Algorithm 7, starts by iterating through each entry n of the given order O (line 2). An entry may either be a task or a stream. For each entry, we determine where it will be scheduled and create an indexable list L with all these locations (line 3). For a task, that set would contain just one end-system, while for a stream, it may contain many links (which are synonymous to an output port of a switch/ES) and also multiple end-systems, if the stream is secure, thus requiring MAC generation/verification.

Using these locations we also create a set of blocks (line 4). A block b is a tuple $(e, l, o, \underline{o}, \overline{o}, prev, next)$ which is associated to an entry e (task/stream) and a location l (node/link). o represents the block offset. $\underline{o}$ and $\overline{o}$ are parameters representing a lower and upper bound on the offset, which are used during the algorithm. The set B is implemented as a linked list, where *prev* and *next* are references to neighboring blocks on the route L. Note that, in the case of multicast streams, *next* could contain references to multiple blocks.

We now iterate over all these blocks (lines 7-8). For each block, we begin by calculating the lower bound on the offset (line 9)[2]. Usually, this lower bound is going to be the end-time (offset+length) of the block on the previous link, making sure that a stream is scheduled consecutively along its route. The first block is the maximum of all end-times of the last blocks of the predecessors of the current entry n in the precedence graph. For example, for application λ_1 in Figure 3.7, the lower bound of the offset of the block of t_3 would be set to the maximum of the end-times of the last blocks of s_1, s_2^0 and s_2^1.

Also, for a secure stream, for all blocks on receiver ESs (i.e., MAC validation tasks), the lower bound is set to the end-time of the corresponding key verification task in the TESLA interval after the stream was received on the ES, since, according to the TESLA security condition, the stream can only be authenticated from that point on.

In the next step, the earliest possible offset for the current block is calculated (line 10). This function returns the earliest offset greater or equal to the lower bound within the feasible region. For more detail, see Section 34.

If such an offset is found, and it is smaller than or equal to the upper bound, we can assign it to the block (line 14). We then iterate through each of the following blocks and set their upper bound to the latest point in time when their node is available and has been since the offset (line 15-18). This is done to fulfill the TSN constraint, which forbids different streams to interleave within a queue (c.f. [RP17], [CSCS16] for a more detailed explanation).

If such an offset is found, but it is larger than the upper bound, it is impossible to schedule the block while the port is still available, i.e., without it interleaving with other streams (line 25). Consequently, we have to backtrack and schedule the previous block at a later time. Therefore, we set the lower bound *of the previous block* to the earliest time when the current port is available and remains so until the offset (line 26-27).

Figure 3.8 gives an example of this process. In step 1, s_1 has already been scheduled, and we are in the process of scheduling s_2. We have scheduled the first block on l_{e_2,sw_1} and are now trying to schedule the second one on l_{sw_1,e_3}. The lower bound of our offset $\underline{o}$ is set to the end-time of the first block. The upper bound $\overline{o}$ is set to the latest time after which l_{sw_1,e_3} is still available after the offset of the first block, i.e., the start time of s_1 on that link. Finally, we find the earliest offset o to be only after the end time of s_1. It cannot be earlier, since then the blocks of s_2 and s_1 would overlap. However, scheduling s_2 at that time is not possible since it would mean that the two streams interleave at the same port. Consequently, in step 2, we backtrack and reschedule the first block of s_2 by setting the lower bound on its offset to the earliest time when its

[2]The algorithm can be found in Section 3.11

Algorithm 7: Scheduling - ASAP Heuristic

```
1  Function Schedule(P, R)
2      foreach n ∈ O do
3          L = GetRoute(n, R);
4          B = CreateBlocks(n, L);
5          l = L[0];
6          i = 0;
7          while true do
8              b = B[l];
9              b.o_ = CalculateLowerBound(n, b, P, R);
10             o = EarliestOffset(b, l);
11             if o == ∞ then
12                 return false;
13             else if o ≤ b.ō then
14                 b.o = o;
15                 foreach g ∈ b.next do
16                     if IsBlockOnLink(g) then
17                         g.ō = LatestQueueAvailableTime(g, o);
18                 end
19                 i = i + 1;
20                 if i < len(L) then
21                     l = L[i];
22                 else
23                     break;
24                 end
25             else
26                 g = b.prev;
27                 g.o_ = EarliestQueueAvailableTime(b, o);
28                 l = b.prev.l;
29                 i = L.indexOf(l);
30             end
31         end
32         Σ = UpdateSchedule(n);
33     end
34     return Σ;
```

port is available and remains so until o. In step 3, we are able to schedule the second block of s_2 without problems.

Once we have successfully found an offset for each block, we can update the schedule (line 32). This will remove the found blocks B from the feasible region.

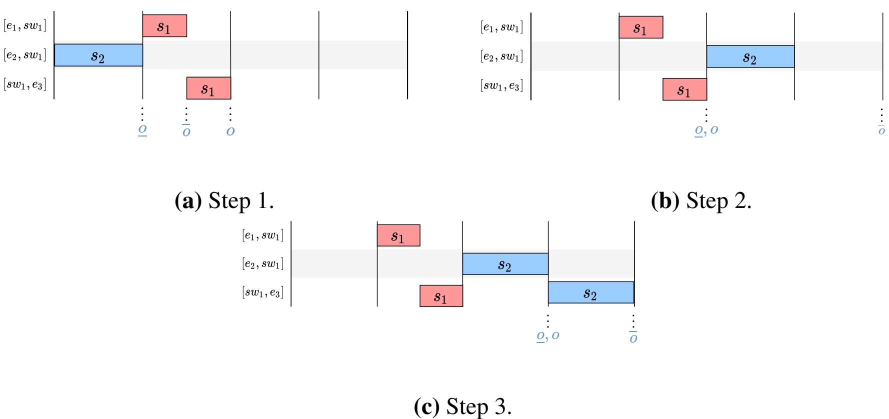


Figure 3.8: Backtrack Example: Scheduling s_2.

Calculating the Earliest Offset

Calculating the earliest offset for a given block is an important part of the heuristic. Algorithm 8 shows the function. It takes a block b as an input and calculates the feasible region for that block (line 2). It then returns the lowest possible time that is within the feasible region and greater or equal than the lower bound (lines 3-6).

Algorithm 8: ASAP Heuristic - EarliestOffset

```
1  Function EarliestOffset(b)
       /* ordered set of intervals */
2      I = GetFeasibleRegion(b);
3      foreach i ∈ I do
4          o = max(b.o̲, i.begin);
5          if i.contains(o) then
6              return o;
7      end
```

The function to calculate the feasible regions for a given block b is detailed in Algorithm 9. We start by getting all free intervals on the node/link $b.l$ for the period $b.e.T$ of the block (line 3). This ensures that the feasible region does not include any previously scheduled blocks on that node/link. The function then proceeds to fill the data structure R_{feas} with the free intervals, while cutting off a piece with the length of the block b from the end of each such interval (lines 4-7). This makes the feasible region represent all feasible values for the *offset* of the block.

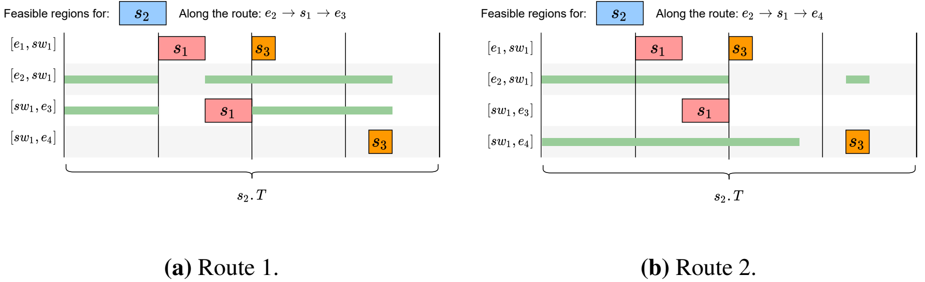
If the block is assigned to a link, we have to cut down the feasible region further. Due

to the TSN isolation constraint, it is not allowed to transmit two different streams on the same port at the same time. Thus, we iterate here over all the subsequent blocks b_{next} of the current block b, i.e., the blocks on the next links/ES on the route of the stream associated with the block (line 9). If the next block is also assigned to a link (not to an ES), we iterate through all already scheduled blocks b_{other} on that link $b_{next}.l$. These are blocks from other streams with whose predecessors, wherever they are scheduled, we are not allowed to overlap. Thus, we cut the interval $(b_{other}.prev.o,\ b_{other}.o)$ from the feasible region (line 13).

Algorithm 9: ASAP Heuristic - GetFeasibleRegion

```
1  Function GetFeasibleRegion(b)
2      R_feas = ∅;
3      B_free = GetFreeIntervals(b.l, b.e.T);
       /* (i) Add all free intervals that could contain block b */
4      foreach iv ∈ B_free do
5          if iv.end − Length(b) ≥ iv.begin then
6              R_feas = AddToFeasibleRegion(R_feas, (iv.begin, iv.end - Length(b)));
7      end
8      if IsLink(b.l) then
           /* (ii) Cut out the interval blocked by other streams on
              the next port (TSN Stream Isolation) */
9          foreach b_next ∈ b.next do
10             if IsLink(b_next.l) then
11                 foreach b_other ∈ GetAllBlocksForLink(b_next.l) do
12                     if b_other ≠ b_next then
13                         R_feas = CutFromFeasibleRegion(R_feas,
                            (b_other.prev.o, b_other.o));
14                 end
15         end
16     return R_feas;
```

Figure 3.9 provides two examples of feasible regions, shown in green, for a stream s_2 on two different routes. Looking at Figure 3.9(a), note the free space at the end of the period and before s_1 on l_{sw_1,e_3}. Choosing an offset anywhere in this space would result in s_2 being scheduled outside its period or overlapping with s_1. Choosing an offset in the first free space on l_{e_2,sw_1} would result in s_1 and s_2 being transmitted to the same port at the same time, breaking the TSN isolation constraint. Note how in Figure 3.9(b) this is not the case, since s_2 is transmitted to a different port (l_{sw_1,e_4}) than s_1.



(a) Route 1. **(b)** Route 2.

Figure 3.9: Feasible Region Example.

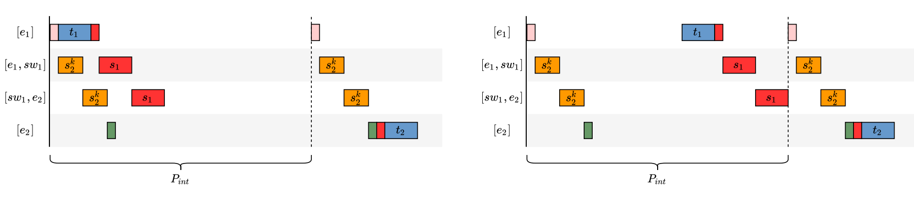
3.7.6 Optimizing the Latency for Secure Streams

After we have created a new schedule, we can apply some post-processing to minimize its latency. Since TESLA requires a separation of sending and receiving tasks into separate intervals and since we are using an ASAP heuristic, there can be a significant gap between those tasks, as can be seen in Figure 3.10(a), resulting in an increased latency. To minimize the latency, the algorithm in Algorithm 10 will go through each secure stream of each application (line 4). It will use the *OptimizeLatencyForSecureStream* function in Algorithm 11 to optimize each stream individually. This function shifts all instances of the given stream as close to the instances on the receiver end-system as possible without breaking the TESLA constraint. It also has an optional boolean parameter. If that is set, it also shifts the sending task of the given stream (otherwise, there would be no latency gain). However, when we are optimizing a redundant stream, i.e., a stream where multiple copies originate at the same task, said task should only be moved when the last copy is optimized (lines 6-11). Otherwise, we can shift it immediately (line 13)

The *OptimizeLatencyForSecureStream* function in Algorithm 11 works internally by looping through the list of receivers of the given stream (line 2, multiple in case of a multicast stream). It goes backwards through the linked list of blocks for the stream, starting with the block on the last link before the current receiver (line 4). For each block, it will increase the offset as much as possible (move them as far as possible to the right) (line 10). After changing the offset, we update the schedule (line 11). Then we continue iterating through the linked list (lines 17-19). If we arrive at the last block and the *move_task* boolean is set, we finish by the offset of the sender task (lines 12-16).

Algorithm 10: ASAP Heuristic - OptimizeLatency

```
1  Function OptimizeLatency(Σ, P)
2      foreach λ in Λ do
3          foreach n ∈ TopologicalOrder(λ, P) do
4              if IsStream(n.e) and n.e.secure and n.e ∈ S^d then
5                  if n.e.rl > 1 then
6                      foreach s_r ∈ S_{n.e} do
7                          if s_r ≠ n.e then
8                              n_r = GetNode(s_r, P);
9                              OptimizeLatencyForStream(n_r, False);
10                     end
11                     OptimizeLatencyForStream(n.e, True);
12                 else
13                     OptimizeLatencyForStream(n, True);
14         end
15     end
```



(a) Non-Optimized Stream. **(b)** Optimized Stream.

Figure 3.10: Example Latency Optimization for Secure Streams.

Algorithm 11: ASAP Heuristic - OptimizeLatencyForStream

```
1  Function OptimizeLatencyForStream(n, move_task)
2      foreach es_recv ∈ receivers(n) do
3          b = BlockOnLink(es_recv, n);
4          b_prev = b.prev;
5          t_kv = GetKeyVerificationTask(n.src, e);
6          b_kv = GetBlock(t_kv);
7          i = GetTESLAIntervalForBlock(b_kv);
8          ub = i ∗ t_kv.T − b_prev.L;
9          while b_prev ≠ ∅ do
10             b_prev.o = min(ub, b_prev.o̅);
11             UpdateSchedule(n);
12             if b_prev.prev == ∅ and move_task and IsLastReceiver(es_recv) then
                   /* Also move the sender task closer to the first
                      block of the stream */
13                 t_sender = GetSenderTask(n);
14                 b_sender = GetBlock(t_sender);
15                 ub = b_prev.o − b_sender.L;
16                 b_prev = t_sender;
17             else if b_prev.prev ≠ ∅ then
18                 ub = b_prev.o − b_prev.prev.L;
19                 b_prev = b_prev.prev;
20         end
21     end
```

3.8 Experimental Results

In this section, we evaluate our two solutions to the formulated problem: The Constraint Programming formulation (referred to as *CP*, described in Section 3.6) and the Simulated Annealing metaheuristic (referred to as *SA*, described in Section 3.7). We analyze their scalability, runtime, and solution quality and evaluate the impact of added redundancy and security.

Both solutions were implemented in Python 3.9. We developed a software tool with a web-based interactive user interface to display the models and solutions, including a routing graph and the schedule.[3] For solving the CP formulation, we use the CP-SAT solver from Google OR-Tools [Goo]. For calculating k-shortest-paths in the metaheuristic we use the *shortest_simple_paths* function from the NetworkX [HSS08] library. All evaluations were run on a High-Performance Computing (HPC) cluster, where each node consists of two Intel Xeon 2660v3 Processors with 10 cores running at 2.60GHz and 16GB memory. Both CP and SA run on one node at a time.

3.8.1 Test Cases

For the scalability evaluation, we used the following test cases, see Table 3.3: the example presented in Section 3.5 (*example*), a realistic automotive test case from a large automotive manufacturer (*auto*) [GZPS17], a medium-sized automotive case study from [KHM05] (*case_study*) and 16 synthetic test cases of increasing size and complexity. The topology of the auto test case was adjusted to allow disjunct redundant routes.

For the redundancy/security impact evaluation, we used an additional set of 100 synthetic test cases grouped into four batches.

We created the synthetic test cases to be as realistic as possible: They all feature secure streams, redundancy levels between 1 and 3, applications with complex dependencies and a realistic network topology that allows disjunct redundant paths.

To create realistic topologies, we developed a custom algorithm, as follows. For a given number of switches and end-systems, we create that many random points in 2D space. Then we connect each switch to its closest neighbor until every switch is connected to 4 other switches. Afterward, we connect each end-system to the closest 3 switches.

To create realistic application DAGs, we used the GGen tool presented in [CMP+10]

[3] The tool including the obtained results is available on GitHub: `https://github.com/nreusch/TSNConf`

and the layer-by-layer method with a depth of 3 and a connection probability of 50%. If a DAG contains separate subgraphs, these are split into separate applications. The application period is chosen randomly among the set {10, 15, 20, 50ms}. Nodes of the generated DAG are interpreted as tasks with a random WCET, upper bound at 6% of the period. Tasks are divided randomly between ES. All outgoing edges of a node in the DAG combined are interpreted as a stream, with the source node as sender task and the destination nodes as receiver tasks. The stream has a random size below or equal to 1.500 Bytes, with a random RL between 1-3 and a 30% probability to be considered security-critical.

We used a link speed of 1000 Mbit/s for all links in the network. TESLA uses 16 B keys and MACs, and a hash computation takes 10 µs on every ES.

3.8.2 Scalability Evaluation

To evaluate the scalability, we ran both the CP and the SA solutions on the same test cases with the same computing resources. Table 3.3 shows the results for each test case for both solutions. The columns **# ES, # SW, # Streams, # Tasks** give the total number of ES, SW, streams, and tasks, respectively. **# Receiver Tasks** gives the total sum of stream receiver tasks (since we consider multicast streams, one stream can have multiple). The **Cost** column gives the total cost of the found solution following the cost function in Algorithm 6. The **T** column shows the total runtime of the solver.

The CP solution was given a timeout of 60 min. If CP failed to find an optimal solution in time, or ran out of memory, we reported Cost and T as empty "/". The SA solution was given a timeout of 10 min (20 min for the largest test case, giant1). We used ParamILS [HHLBS09] to optimize the following parameters for the SA heuristic: T_{start}, α, k, p_{rmv} and w. a was set to 50000, b to 10000.

Note that the CP solver will return once the optimal solution is found, while the SA solver will always run until the timeout and return the best feasible (i.e., no missed deadlines or overlap) solution found up to that point. However, SA is able to find a first feasible solution very quickly. For all test cases in Table 3.3 it could find one in less than 10 s.

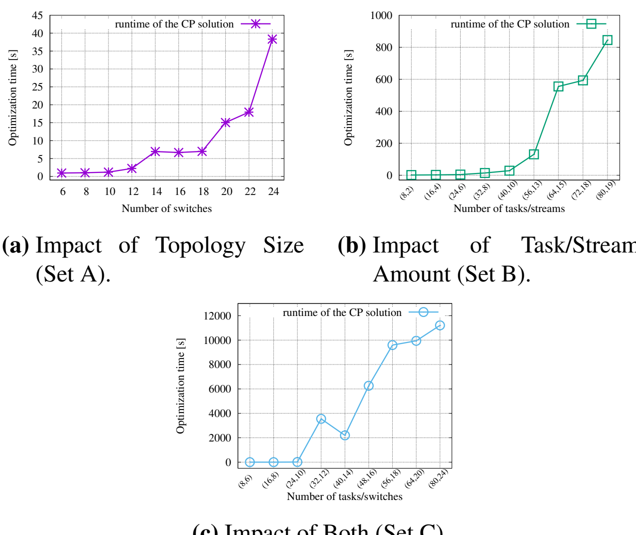
The table shows that CP is able to find solutions up to medium-sized test cases within the given timeout, but it does not scale to the larger test cases. SA is scalable; it is able to find solutions even for the largest test cases. This scalability comes at an increase in cost by 67% on average, which can be reduced by giving a longer timeout. This increase is mostly caused by increased application latencies (scheduling cost), which are still within the deadlines, while the routing cost is usually close to or equal to the optimal routing cost from the CP solution. The conclusion is that SA can be success-

Test case	Method	# ES	# SW	# Streams	# Recv. Tasks	# Tasks	Cost	T
example	CP	4	2	6	10	9	467	1 s
example	SA	4	2	6	10	9	477	10 m
auto	CP	20	32	84	102	74	/	/
auto	SA	20	32	84	102	74	38 031	10 m
case_study	CP	6	2	29	31	28	3 771	130 s
case_study	SA	6	2	29	31	28	6 114	10 m
tiny1	CP	4	2	2	2	6	1 708	0.2 s
tiny1	SA	4	2	2	2	6	1 708	10 m
tiny2	CP	4	2	3	4	6	1 732	0.2 s
tiny2	SA	4	2	3	4	6	1 732	10 m
tiny3	CP	4	2	11	13	15	7 450	14 m
tiny3	SA	4	2	11	13	15	18 088	10 m
small1	CP	8	4	10	16	20	5 421	2 s
small1	SA	8	4	10	16	20	13 303	10 m
small2	CP	8	4	14	20	23	9 110	60 m
small2	SA	8	4	14	20	23	13 794	10 m
small3	CP	8	4	29	48	35	7 705	17.5 m
small3	SA	8	4	29	48	35	13 781	10 m
medium1	CP	16	8	23	34	37	12 991	4.5 m
medium1	SA	16	8	23	34	37	22 883	10 m
medium2	CP	16	8	30	47	43	6 552	5.2 m
medium2	SA	16	8	30	47	43	19 455	10 m
medium3	CP	16	8	36	53	47	15 515	60 m
medium3	SA	16	8	36	53	47	26 486	10 m
large1	CP	32	16	47	86	73	/	/
large1	SA	32	16	47	86	73	43 872	10 m
large2	CP	32	16	33	65	72	24 953	25 m
large2	SA	32	16	33	65	72	41 026	10 m
large3	CP	32	16	69	170	104	/	/
large3	SA	32	16	69	170	104	34 860	10 m
huge1	CP	64	32	84	183	133	/	/
huge1	SA	64	32	84	183	133	73 070	10 m
huge2	CP	64	32	99	213	161	/	/
huge2	SA	64	32	99	213	161	57 246	10 m
huge3	CP	64	32	99	197	169	/	/
huge3	SA	64	32	99	197	169	93 357	10 m
giant1	CP	128	64	144	347	261	/	/
giant1	SA	128	64	144	347	261	101 799	20 m

Table 3.3: Results of Scalability Tests.

fully used to route and schedule large realistic test cases, and its quality is comparable to the optimal solutions obtained by CP.

Furthermore, we have investigated the impact of scaling different parameters on the runtime of the CP solution. We created three sets of test cases: The set **A**, used in Figure 3.11a, in which we keep a fixed set of applications but scale the size of the network topology, the set **B**, used in Figure 3.11b, in which we keep a fixed network topology and scale the number of tasks, and streams and the set **C**, used in Figure 3.11c in which



(a) Impact of Topology Size (Set A).

(b) Impact of Task/Stream Amount (Set B).

(c) Impact of Both (Set C).

Figure 3.11: Scalability Results of CP Solution.

we scale both at the same time. We ran the tests on the same computing resources and took the average runtime of multiple runs to reduce the impact of external factors (disk access, caching, etc.). While, as expected, all the parameters have a measurable impact on runtime, we noticed that the number of streams and the complexities of their routes have a significant impact, as can also be seen in Figure 3.11b.

3.8.3 Impact of Adding Redundancy and Security to a Test Case

Fulfilling the security and redundancy requirements of applications introduces extra tasks and streams that need to be routed and scheduled, leading to an overhead compared to ignoring those security and redundancy requirements. In this set of experiments, we were interested in evaluating the overhead of fulfilling the redundancy and security requirements compared to the case these are ignored. These overheads were measured on solution cost, available bandwidth, and CPU resources. Hence, we created four batches of 25 synthetic test cases each. Each test case has a random topology with 8 switches and 16 end-systems, 24 tasks, and multiple applications with random DAGs. Streams have a random RL between 1 and 3 and 30% probability to be considered security-critical. We also ran the same experiments with the auto test case from Table 3.3 to give an expectation for the overhead in a realistic scenario.

Each batch features either large (1 000-1 500 B) or small (1-250B) streams and either

	Batch name	Security	Redundancy	Cost	Bandwidth	CPU
0	batch0 - large streams, small tasks	no	no	3822.08	0.09	1.44
1		no	yes	+1.34%	+25.45%	+0.00%
2		yes	no	+256.25%	+5.49%	+12.99%
3		yes	yes	+258.10%	+36.64%	+15.93%
4	batch1 - large streams, large tasks	no	no	17721.32	0.08	7.06
5		no	yes	+-0.00%	+15.91%	+0.00%
6		yes	no	+64.31%	+3.18%	+1.39%
7		yes	yes	+64.98%	+22.16%	+1.75%
8	batch2 - small streams, large tasks	no	no	18547.6	0.01	7.51
9		no	yes	+0.06%	+31.29%	+0.00%
10		yes	no	+52.86%	+33.92%	+0.21%
11		yes	yes	+53.12%	+97.96%	+0.70%
12	batch3 - small streams, small tasks	no	no	3713.32	0.01	1.55
13		no	yes	+0.37%	+26.62%	+0.00%
14		yes	no	+238.77%	+27.73%	+8.54%
15		yes	yes	+245.32%	+85.57%	+10.57%

Table 3.4: Impact of Security and Redundancy Measures.

large ($\leq$10% of period) or small ($\leq$2% of period) tasks. Each batch was run 4 times using the first feasible SA solution with different combinations of enabled/disabled security and redundancy requirements. Disabled security means that all streams are set to a security level of 0, while disabled redundancy means that all streams are set to a redundancy level of 1.

Table 3.4 shows the results. We always take the results for the no-security, no-redundancy run as a baseline and note the percentual increase in total cost, total bandwidth occupation percentage and total CPU utilization percentage in the following rows. Bandwidth and CPU utilization are measured as the mean of the utilization over all links and ESs, respectively. Bandwidth utilization for a specific link is the percentage of the hyperperiod in which streams are traversing this link. CPU utilization for a specific ES is the percentage of the hyperperiod in which tasks are running.

As can be seen, the impact of adding security and redundancy differs significantly, depending on the size of initial streams and tasks. Note that an increase in overhead is expected with an increase in the number and difficulty of the security and redundancy requirements.

Adding redundancy has a negligible impact on cost and CPU utilization, but always has a significant impact on bandwidth. Adding security always has a significant impact on cost. This is because of the fundamental requirement of TESLA that sender and receiver tasks of secure streams lie in separate intervals, which will increase the end-to-end latency of the corresponding application. Therefore, the more consecutive secure streams exist in an application, the higher is the impact. However, our approach will always make sure that, if feasible, no application deadline is missed, despite the higher latency. The impact of adding security on bandwidth and CPU utilization depends

largely on the relative size of streams and WCET of tasks compared to the TESLA overhead. For example, the 16 B of overhead for a MAC is much more significant for a 100 B stream than for a 1 000 B stream.

3.8.4 Discussion

Our proposed SA implementation is able to determine good solutions in a reasonable time, even for large test cases. In addition, it can find feasible solutions (where all timing, safety, and redundancy requirements are satisfied) extremely quickly, within 10 s even for large test cases. This can be useful, e.g., for evaluating several architectures in terms of their monetary costs and redundancy allowed by the physical topology, prototyping or for rapid runtime reconfiguration in case of failures or changes in traffic patterns. Although CP can find optimal solutions, it does not scale for large test cases, and it is not flexible; that is, it will not report solutions that are not feasible. An advantage of SA is its ability to find return near-feasible solutions for those test cases that cannot be solved, i.e., solutions with some infeasible applications or overlapping streams. SA can point out the offending apps/tasks and streams, which can give hints of where the configuration has to be improved to become feasible, e.g., by increasing the redundancy in the physical topology or by changing the mapping of tasks to ESs.

3.9 Conclusion

In this paper, we addressed the combined TSN routing and scheduling problem for complex applications with redundancy and security requirements. We used TESLA, which is an efficient authentication protocol for use-cases with multicast communication between low-power devices, with a modification to the protocol that makes it more lightweight, made possible by the real-time requirements of our network.

We developed two methods to solve the combined routing and scheduling problem: A Constraint Programming solution that can solve small and medium-sized test cases optimally and a solution that combines a Simulated Annealing metaheuristic and an ASAP list scheduling that can solve very large test cases. In the process, we formalized the constraints governing our problem and introduced novel ways to handle the complexities introduced by TESLA and redundancy while calculating correct solutions in the heuristic. Additionally, we developed an open-source tool for the reuse of our solutions and interactive visualization of routes and schedules. Finally, we evaluated the impact of adding security and redundancy to existing applications and showed that the majority of overheads depend on the size of existing tasks and streams and thus ultimately on the requirements of the application area.

3.10 Appendix A: Routing Constraint Formulation for Allowed Overlap

To achieve a constraint formulation in which overlap of redundant streams is possible, the following steps need to be done:

Replace (RC2):

$$cost(s_n) = length_cost(s_n) + 100 * overlap_cost(s_n) \tag{RC2}$$

Introduce the following:

$$length_cost(s_n) = \sum_{n \in \mathcal{N} \setminus \{s_n.t_s.e\}} (x(s_n, n)! = nil) \tag{RC3}$$

$$overlap_cost(s_n) = \sum_{n \in \mathcal{N} \setminus \{s_n.t_s.e\}} \sum_{m \in \mathcal{N} \setminus \{n\}} link_cost(s_n, n, m) \tag{RC4}$$

$$link_cost(s_n, n, m) = (xsum(s_n, n, m) - 1) * (x(s_n, n) == m) \tag{RC5}$$

Remove (R6). The overlap cost for a stream is dependent on how many redundant copies a stream overlaps with, and on how many links. In (RC2) the total overlap cost is weighted with 100, but other values are also possible.

3.11 Appendix B: More Functions from Metaheuristic Formulation

3.11.1 CalculateLowerBound

The function in Algorithm 12 determines the lower bound on the offset of a given block b. For a task or first stream instance, this is the maximum end-time of the blocks of the predecessors of the current entry n in the precedence graph P (lines 3–7). This is the maximum end-time of all blocks from predecessor links for other stream instances on links (lines 8-12). Finally, for the stream instance on a receiver end-system, which models a MAC validation, the lower bound is the end-time of the corresponding key verification task, which is necessary to happen before a MAC can be validated (lines 13–17).

Algorithm 12: ASAP Heuristic - CalculateLowerBound

```
1  Function CalculateLowerBound(n, b, P, R)
2      lb = 0;
3      if b.prev == ∅ then
           /* If n is a task or the first stream instance */
4          foreach n_prev ∈ Predecessors(n, P) do
5              b = LastBlock(n_prev);
6              lb = max(lb, b.o + Length(b));
7          end
8      else if IsLink(b.l) then
           /* If n is a stream and b.l is a link */
9          foreach l_prev ∈ PredecessorLinks(b.l, n, R) do
10             b_prev = BlockOnLink(b.l, n);
11             lb = max(lb, b_prev.o + Length(b_prev));
12         end
13     else
           /* If n is a stream and l is a receiver end-system */
14         t_key^verify = GetKeyVerificationTask(n, l);
15         b_key^verify = GetBlockForEntry(t_key^verify);
16         i = GetTESLAIntervalForBlock(b_key^verify);
17         lb = b_key^verify.o + i * b_key^verify.e.T + Length(b_key^verify);
18     end
19     return max(lb, b.o̲);
```

3.11.2 UpdateSchedule

The function in Algorithm 13 updates the schedule with all the blocks that were calculated for a given entry n. It iterates through all the links/nodes the entry is scheduled on (line 3). It calculates all offsets and end-times across the hyperperiod (lines 4–8) and proceeds to cut out the appropriate parts from the feasible regions for each existing period in the network (lines 9–20).

Algorithm 13: ASAP Heuristic - UpdateSchedule

```
1  Function UpdateSchedule(n)
2      L = GetLinks(n);
3      foreach l ∈ L do
4          for i = 0 to H/(n.T) do
5              offsets[i] = b.o + i * b.e.T;
6              endtimes[i] = b.o + i * b.e.T + Length(b);
7              i = i + 1;
8          end
           /* Block queues/end-systems */
9          foreach T ∈ Periods do
10             for i = 0 to len(offsets)-1 do
11                 o = offset[i];
12                 e = endtimes[i];
13                 if e%T < o%T then
                       /* Handle wrap around period border */
14                     CutFromFeasibleRegion(T, l, o%T, T);
15                     CutFromFeasibleRegion(T, l, 0, e%T);
16                 else
17                     CutFromFeasibleRegion(T, l, o%T, e%T);
18                 end
19             end
20         end
21     end
```

CHAPTER 4

Paper C: Mapping and Scheduling Real-Time Applications on Edge Computing Platforms with Remote Attestation for Security

---

Edge Computing Platforms (ECP) increasingly have to integrate diverse applications with mixed-criticality requirements. In this paper, we consider that critical applications and non-critical Edge applications share an ECP. Critical applications are implemented as periodic hard real-time tasks and messages and have stringent timing and security requirements. Edge applications are implemented as aperiodic tasks and messages, and are not critical. We assume that the critical tasks are scheduled using static cyclic scheduling. Time-Sensitive Networking (TSN) is used for dependable communication, and Remote Attestation (RA) is employed to check that the platform components are secure. We are interested to determine an optimized mapping and scheduling of critical and Edge applications, such that (i) the deadlines of the critical applications are guar-

anteed at design-time, (ii) the platform has resources to perform RA, and (iii) we can successfully accommodate multiple dynamic responsive Edge applications at runtime. We evaluate our approach on a realistic use case. The results show that our approach generates dependable solutions that can meet the timing constraints of the critical applications, have enough periodic slack to perform RA for security, and can accommodate Edge applications with a shorter response time.

4.1 Introduction

We are at the beginning of a new industrial revolution (Industry 4.0), which will bring increased productivity and flexibility, mass customization, reduced time to market, improved product quality, innovations, and new business models [LFK+14]. However, Industry 4.0 will only become a reality through the convergence of Operational and Information Technologies (OT & IT). OT consists of cyber-physical systems that monitor and control physical processes that manage, e.g., automated manufacturing, critical infrastructures, smart buildings, and smart cities. These application areas are typically safety-critical and real-time, requiring guaranteed extra-functional properties, such as, real-time behavior, reliability, availability, safety, and security, and are often required to show compliance to industry-specific standards.

Edge Computing is envisioned as an architectural means to realize the IT/OT convergence [PZB+21]. It is a new architectural paradigm in which the resources of an edge server are placed at the edge of the Internet, in proximity to cyber-physical systems, mobile devices, sensors and IoT endpoints [SPX19]. With Edge Computing, devices are extended with computational and storage resources to enable a variety of communication and computation options, which will lead to improved interoperability, security, more efficient and rich control, and higher manufacturing efficiency and flexibility. Several initiatives are currently working towards realizing this vision [PML+19, PZB+21].

Mixed-criticality applications can be classified in several ways depending, e.g., on their safety-criticality and time-criticality. In safety-critical systems, a failure (e.g., due to a malfunction or a security attack) may lead to loss of life or damage to the environment or property. Many safety-critical systems are also real-time, where the correctness of the results depends also on the time when they are delivered, e.g., deadlines have to be satisfied. Conversely, noncritical dynamic applications are not safety related and do not have stringent timing requirements. To realize the vision of Industry 4.0, such non-critical dynamic applications e.g., related to new business models, data analytics, software updates for security and maintenance, and connected equipment services, have to be hosted by an Edge Computing Platform (ECP). These Edge applications are aperiodic, i.e., their arrival-time is unknown, and the ECP should dynamically allocate

resources such that their quality-of-service is maximized, e.g., their response times are reduced. To enable this, we have to map and schedule the critical applications at design time in a way that leaves periodic slack in the schedule table, to be used by Edge applications.

An Edge Computing Platform (ECP) includes Edge Devices (EDs) capable of communicating and executing computations in the proximity of the "things" (sensors, actuators, IoT endpoints, etc.) and data sources to guarantee effective collaboration between devices, nodes, and the cloud. An ED is a compute node that integrates mixed-criticality applications that share the ECP. Regarding computation, we assume that the critical tasks are running in a Real-Time Operating System using real-time scheduling policies. We consider static-cyclic scheduling in this paper, as it is suitable for applications of high-criticality [But11]. We assume that mixed-criticality applications can be separated in different *partitions* enforced using hardware-supported virtualization, based on hypervisors, such as ACRN or PikeOS, see [PZB+21, BCP20] for a discussion and references.

Regarding communication, we consider the ECP to use IEEE 802.1 TSN [Ins16c] as the wired communication solution, as envisioned by several industrial consortiums. TSN [Ins16c], which is becoming the standard for communication in several application areas, e.g., from industrial control to automotive and aerospace, is a set of amendments to the IEEE 802.1 standards [Ins16c], equipping Ethernet with the capabilities to handle real-time mixed-criticality traffic with high bandwidth. TSN supports multiple traffic types, and hence, is suitable for mixed-criticality applications running on an ECP. Researchers and standardization bodies are also working towards extending TSN capabilities over wireless networks (e.g., IEEE 802.11 and 5G). The solution presented in the paper can also be extended to consider wireless TSN.

As ECPs become more interconnected with the outside world, new attack vectors are possible [PBA17, XJL+19] that may also compromise safety. Therefore, we also consider security aspects in our work. One desirable goal in large heterogeneous ECP's is to assure the integrity of devices. Malicious behavior of devices should be detected in a timely manner and appropriate measures taken, e.g., the filling of a tank should be stopped if the pressure sensor is found to report fake values. One promising direction is to use *Remote Attestation* (RA) to authenticate the hardware and software configuration of a remote device, thus allowing the provision of strong assurance guarantees.

RA provides a mechanism to validate the integrity of software running on untrusted devices. The assumption is that there are both trusted (e.g., high-end EDs) and untrusted devices (e.g., low-end IoT endpoints) in our network. For validation, the trusted party, called verifier, sends an attestation request to an untrusted party, called prover. The prover responds with a certificate of its currently running software, e.g., by including a hash of its memory content. Measures are in place such that that certificate cannot be faked by an attacker controlling the prover or the communication link. The verifier

checks this certificate and that the memory content is as expected. If that is not the case or no response is received to the attestation request, the verifier initiates appropriate measures (out of scope for this work) [SL16].

4.1.1 Related Work

This work touches on a lot of different configuration problems typical for safety-critical systems: task mapping, task and message scheduling, routing, and security are all actively studied topics.

The task mapping problem is concerned with assigning tasks to different, possibly multicore, devices in the network [Sin06]. Typical goals are to meet real-time deadlines and fulfill computational or energy resource constraints [GBI21, SP18]. Another goal, which becomes increasingly relevant in heterogeneous networks with devices of significantly different computational capabilities, is offloading of tasks from less to more powerful devices [RZH+19, SGAS20]. This allows for a reduction in energy consumption and execution times [CTHC15, DTF16]. Some work has looked at mapping tasks of mixed-criticality on the same platforms [BD17, TSP15, BP22]. This comes with the benefit of, for example, savings in cost, weight, space and energy, but requires careful partitioning mechanisms to avoid harmful interaction between tasks.

Task scheduling in real-time systems is a very well-studied topic, and researchers have considered various scheduling policies, from non-preemptive static-cyclic scheduling, considered in this paper, to preemptive dynamic scheduling [But11]. Researchers have also addressed the problem of real-time message scheduling [Ste10]. Many works consider scheduling in TSN using the Time-Aware Shaper (TAS), which requires the synthesis of GCLs [CSCS16, PLCS16]. Although initially the work has focused only on message scheduling, recent work proposes solutions to extended problems, considering task scheduling and routing [Bar21, RCP22].

Security in Edge/Fog Computing and IoT has been extensively investigated [TDDFD20]. There is also some work specifically focusing on real-time systems. The authors in [ZZJ+13] show a tradeoff between better security and task execution time/energy. In [JPJ17] the authors consider the overhead for secure communication between tasks on different devices, which scales with the strength of the encryption. Remote Attestation (RA) is a promising approach to authenticate untrusted remote devices in heterogeneous networks [SL16]. RA has been studied in the context of IoT and Fog/Edge Computing [ADD21], but the timeliness aspects of RA, e.g., that it may lead to deadline misses for real-time applications, have been ignored.

Researchers have investigated the impact of security mechanisms on safety-critical real-time systems, e.g., for resource authentication protocols such as TESLA [ZQLY19,

RCP22] or for protecting messages with cryptography [JEP12, JPJ17]. However, no approaches exist that evaluate the impact of remote attestation on real-time applications, which are increasingly being implemented using Edge Computing platforms.

4.1.2 Contributions

As discussed in Section 4.1.1, researchers have addressed the scheduling of safety-critical real-time applications on ECPs that consider TSN. However, the existing work often looks at computation and communication separately or does not consider mixed-criticality applications or security. In our work, we are interested in schedules that can integrate both critical and Edge applications, i.e., the deadlines of the critical applications are guaranteed and at the same time, the slack in the schedule (the idle times on processors and links) can successfully accommodate multiple dynamic responsive Edge applications at runtime. We consider that we have devices with vastly different computational capabilities and try to optimize the mapping of tasks of critical applications to reduce their latency.

Regarding security, we consider Remote Attestation as a scheme to provide device integrity. We assume that we have trusted end-systems acting as verifiers and untrusted end-systems acting as provers. Critical applications, TSN and RA all place constraints on the scheduling of tasks and messages in the network. Our solution synthesizes schedules such that there is enough periodic slack to run regular attestation to secure the ECP. RA requires regular attestation on prover end-systems to minimize the chance of malware going undetected. We formalize the problem and provide a CP formulation to solve it.

To the best of our knowledge, this is the first work that considers the impact of RA on the implementation of mixed-criticality applications on Edge Computing Platforms.

This paper is organized as follows. In Section 4.2 we introduce the model of the Edge Computing Platform we are considering and explain our usage of TSN and RA. In Section 4.3 we introduce the application model. The problem is formulated in Section 4.4 and our CP solution is presented in Section 4.5. In Section 4.6 we evaluate a solution to a simplified problem on a use case. Section 4.7 concludes the paper.

4.2 Edge Computing Platform Model

The ECP is modeled as a directed graph $\mathcal{G} = \{\mathcal{N}, \mathcal{L}\}$, where $\mathcal{N} = \mathcal{E} \bigcup \mathcal{V} \bigcup \mathcal{P} \bigcup \mathcal{SW}$ is the set of nodes and $\mathcal{L}$ is the set of links. A node can either be an ordinary edge

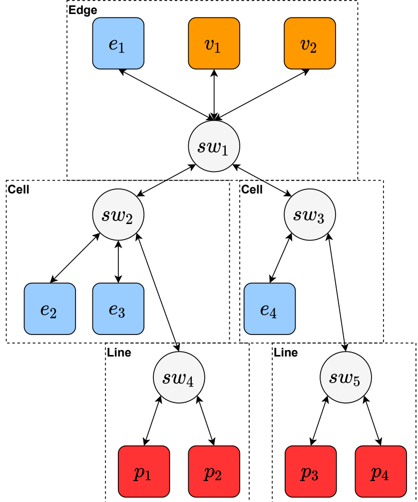


Figure 4.1: Architecture of the Edge Computing Platform.

device $e_i \in \mathcal{E}$, a trusted verifier device $v_i \in \mathcal{V}$, an untrusted prover device $p_i \in \mathcal{P}$ or a network switch $sw_i \in \mathcal{SW}$. A node has an associated capacity $n.C$ which is the percentage of its hyperperiod that can be occupied by critical tasks. The set of links $\mathcal{L}$ represents bidirectional full-duplex physical links. Each link $l_{i,j}$ between nodes n_i and n_j is characterized by the tuple $\langle s, d \rangle$ denoting the speed of the link in Mbit/s and the propagation delay in ms. An example ECP architecture graph is shown in Figure 4.1. The topology is inspired by an industrial use case, in which the network consists of one Edge area, which is connected to production cells containing production lines in a tree-like structure and is connected to the cloud. The devices at the top of the tree are the most powerful in terms of computation power, while the ones at the bottom are the least powerful. Verifiers are assumed to be powerful devices from the edge area, while provers are low-end and exposed devices in production cells, though this could differ from use case to use case.

Mixed-criticality applications, running on edge devices (possibly including verifiers), require different scheduling policies depending on their timing criticality [But11]. Similar to related work, we use static cyclic scheduling (timeline scheduling) for critical control applications. A static cyclic schedule captures the start and finishing time of tasks and flows and repeats with a hyperperiod H, which is the least common multiple of the application periods.

Researchers have used "fixed priority servers" to integrated periodic and aperiodic applications, which run in the slack of the critical application schedules. Such servers are implemented as a periodic task $D_{\mathcal{C}_i}$ that runs in a core $\mathcal{C}_i$ and it is characterized by the

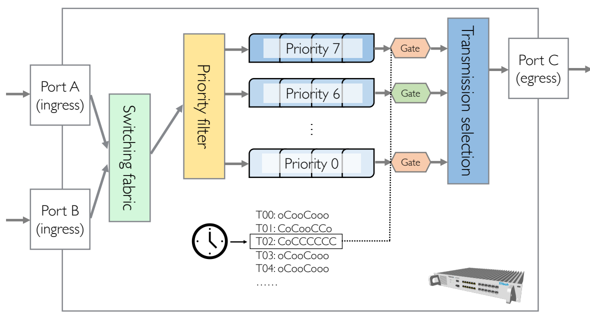


Figure 4.2: TSN Switch.

tuple $\langle c, t \rangle$ denoting the server capacity and the server period in ms. Within a hyperperiod H, a server will have several instances, which are referred to as server slices. Hence, we consider that Edge applications are scheduled using a special type of server called a *deferrable server* [But11]. Deferrable servers use a soft resource reservation technique to allocate its resources to the Edge tasks. Edge application flows are transmitted using the TSN mechanisms specific for strict priority (SP) flows in the windows when ST critical flows are not scheduled, see Section 4.2.1.

At runtime, the ECP handles Edge applications through monitoring and resource management techniques [PZB+21]. Once the Edge applications are submitted to run on the ECP, the *controller* ED, which is determined at run-time using mechanisms such as [KP17], receives the submission request. The controller has knowledge on the available resources on the EDs and SWs. It then decides the placement of Edge application tasks on the EDs using resource allocation techniques, e.g., [SKR+18] which determines the ED that provides the minimum response time for the Edge application.

4.2.1 Time-Sensitive Networking (TSN)

The fundamental mechanisms that enable deterministic temporal behavior over Ethernet are, on the one hand, the clock synchronization protocol defined in IEEE 802.1AS-rev, which provides a common clock reference with bounded deviation for all nodes in the network, and on the other hand, the timed-gate functionality (IEEE 802.1Qbv) enhancing the transmission selection on egress ports. We detail the Time-Aware Shaper (TAS) mechanism defined in IEEE 802.1Qbv in Figure 4.2. The TAS is associated with each traffic class queue and positioned before the transmission selection algorithm. A timed-gate can be either in an *open (o)* or *closed (C)* state. When the gate is open, traffic from the respected queue is allowed to be transmitted, while a closed gate will not allow the respective queue to be selected for transmission, even if the queue is not

empty. The state of the queues is encoded in a local schedule called the Gate-Control List (GCL). Our optimization strategy derives these GCLs.

4.2.2 Remote Attestation

Remote Attestation on our platform is performed as follows: A trusted verifier node v_i sends an attestation request to an untrusted prover p_j. In response, p_j invokes some trusted attestation code $\mathcal{A}ttC$ that measures a region of memory. This measurement is protected using a Message Authentication Code (MAC) with a secret, shared, key K and returned to v_i, which determines whether p_j is in a healthy or compromised state.

The attestation architecture we use is called SMART [EDPT12]. It is a hybrid attestation architecture, suitable for low-end low-powered prover devices. In SMART $\mathcal{A}ttC$ and K are stored in read-only memory (ROM). The key is guarded by MCU access control rules, such that only $\mathcal{A}ttC$ can read it. The execution of $\mathcal{A}ttC$ is non-interruptible and does not leak information. More details about the security assumptions can be found in [EDPT12]. Furthermore, we assume that SMART is extended with a reliable read-only clock (RROC) as proposed in [BRST16], to prevent denial of service attacks on the prover.

In the basic version of SMART, $\mathcal{A}ttC$ attests the whole memory at once, in an uninterruptible process. This is infeasible for our platform, since this process would take multiple seconds on low-powered devices, in which no critical real-time task could be executed. Instead, we adopt the ideas of SMARM [CRT18]. This is a technique that is built on top of SMART. However, instead of attesting the whole memory, the memory is divided into blocks $M_1, ..., M_n$ of size BS. For a given attestation request, only a certain block M_i is attested, whereby i is randomly determined based on the attestation request. Depending on the frequency and block size, this makes it improbable for malware to hide, given that it cannot predict the memory location that is going to be attested. From the scheduling side, we can choose these parameters appropriately, such that we have a good attestation coverage while also guaranteeing all deadlines of critical tasks. We will choose a block size, such that all remaining slack, after scheduling critical tasks, is used on attestation.

4.3 Application Models

Our application model consists of (i) a set of critical applications considered at design-time, denoted with Λ^{crit}, which we capture using a periodic hard real-time task model, (ii) a set of Edge applications considered at runtime, denoted with Λ^{edge}, for which we

use an aperiodic best-effort task model and (iii) a set of Remote Attestation applications Λ^{ra} which are generated at design-time after the scheduling of critical applications.

4.3.1 Critical Application Model

Critical applications appear periodically and consist of tasks and streams. Each critical application $\lambda_i \in \Lambda^{crit}$ is modeled with a Directed Acyclic Graph (DAG), where nodes represent tasks and edges represent data flows between the nodes, see, for example, Figure 4.3. The set of all tasks and the set of all streams in a critical control application are denoted with $\lambda_i.\mathcal{T}$ and $\lambda_i.\mathcal{S}$ respectively. The set of all tasks is denoted as $\mathcal{T}$, the set of all streams as $\mathcal{S}$.

A critical task $t_m \in \lambda_i.\mathcal{T}$ is characterized by the tuple $\langle T, e \rangle$ denoting the task period and the device the task is mapped to. The Worst-case Execution Time (WCET) of a task depends on the device it is mapped to. We define a function $\omega : \mathcal{T} \times \mathcal{N} \rightarrow \mathbb{N}$ that maps any tuple of task $t \in \mathcal{T}$ and device $n \in \mathcal{N}$ to the worst-case execution time of t on n. In some use cases there might be a constraint on which device a task may be mapped. The task deadline is equal to its period. Each task is ready to execute when all its inputs have arrived. The output of a task is produced upon the termination of the task. The task t_m will have $H/t_m.T$ instances denoted with $|t_m|$ in a hyperperiod H which are referred to as jobs denoted with t_m^j. A job is associated with ϕ denoting the start time of the job.

A stream $s_m \in \lambda_i.\mathcal{S}$ is responsible for sending the frames that encapsulate the data from an application, and it is characterized by the tuple $\langle p, b, T, t_s, T_d \rangle$ denoting the priority, the size in bytes, the period, sending task and set of receiving tasks. The deadline, i.e., the maximum allowed end-to-end delay, is equal to this period. A stream may be multicast, i.e. have multiple receivers. The frames of a stream will have to be

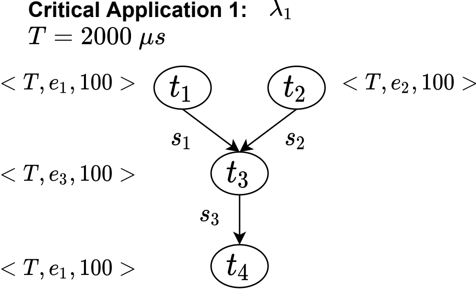


Figure 4.3: Critical Application Example.

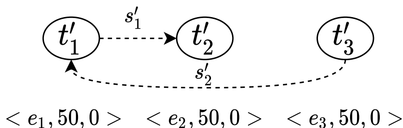


Figure 4.4: Edge Application Example.

transmitted multiple times within a hyperperiod, and we refer to each such transmission as an *instance* of a stream. The number of instances for a stream s_m is denoted with $|s_m|$, and is derived from the period of the stream T and the hyperperiod H.

Each stream s_m is transmitted via a route $\mathcal{R}^{s_m}$ which is represented by ordered list of devices and switches, e.g., $\mathcal{R}^{s_m} = [e_1, sw_1, sw_2, e_2]$. We assume that each stream is associated to only one route, but several streams may share the same route.

4.3.2 Edge Application Model

Each Edge application $\lambda'_i \in \Lambda^{edge}$ consists of a set of aperiodic tasks and a set of aperiodic streams denoted by $\lambda'_i.\mathcal{T}$ and $\lambda'_i.\mathcal{S}$, respectively. The tasks do not have data dependencies, i.e., a task will start when it arrives, but they may exchange data asynchronously using streams. An Edge task $t'_n \in \lambda'_i.\mathcal{T}$ is denoted by the tuple $\langle e, w, a \rangle$ denoting the node it is mapped to, the workload (which is the average execution time) in µs, and its arrival time in µs. The arrival times, mapping and workloads of Edge tasks are unknown at design time. An Edge stream $s'_n \in \lambda'_i.\mathcal{S}$ is also aperiodic. Such a stream is denoted by the tuple $\langle p, b, a \rangle$ denoting the priority, size in bytes, and the arrival time in µs. Figure 4.4 shows an example edge application.

4.3.3 Remote Attestation Model

A Remote Attestation application follows the same model as critical applications, but always has the same Direct Acyclic Graph (DAG) and is scheduled after critical applications during the design phase. The RA application $\lambda^*_i \in \lambda^{ra}$ consists of three tasks t_{v1}, t_p, $t_{v2} \in \lambda^*_i.\mathcal{T}$ and two streams $s_{req}, s_{res} \in \lambda^*_i.\mathcal{S}$. t_{v1} is executed on a verifier and emits the stream s_{req} containing the attestation request. t_p is executed on a prover in response to this request and emits the stream s_{res}. All tasks and streams are scheduled in the slack leftover after scheduling critical applications. Figure 4.5 shows an example RA application.

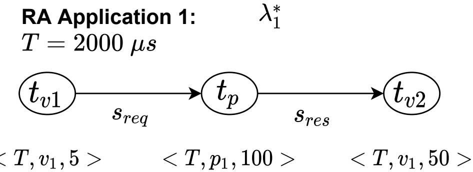


Figure 4.5: RA Application Example.

4.4 Problem Formulation

We formally define the mapping and scheduling problem we address in the paper as follows. Given (1) an ECP modeled with an architecture graph $\mathcal{G}$, (2) a set of critical real-time applications Λ^{crit}, and (3) a set of verifier/prover pairs, we want to determine a configuration Ψ consisting of: (i) the mapping of tasks to end-systems, (ii) the static task schedule tables, (iii) the routes for each stream, (iv) the GCLs for each switch, (v) the period and capacity of the deferrable servers D_{C_i} on edge devices, and (vi) the RA applications Λ^{ra}.

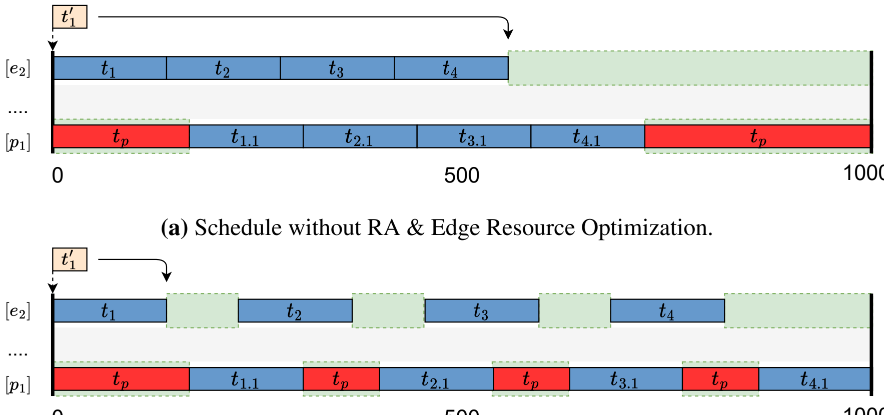
Although not part of the optimization problem definition, note that at runtime, our approach handles (vii) the migration of Edge applications to the nodes that have resources for their execution, (viii) the scheduling of Edge tasks on the servers and of their streams on TSN.

We are interested in an optimized configuration Ψ such that: the deadlines of all the critical applications are met and the resources available for the RA and Edge applications are maximized, such that we can maximize the RA-based security and minimize the response times for Edge applications. The quality of a configuration is determined by the cost function Equation CS1 introduced in Section 4.5.

Synthesizing a static task schedule for the nodes is equivalent to determining the task mapping $t_m.e$, the offsets $t_m^j.\phi$, and the server slices' offsets $D_{C_i}^j.\phi$. Additionally, synthesizing GCLs for the ports of network nodes is equivalent to determining the critical streams' routes $s_m.r$ and offsets $s_{m,n}^k.\phi$.

4.4.1 Example

Let's consider an example using the architecture in Figure 4.1. This network architecture may belong to a factory with an edge computing center, containing powerful machines acting as verifiers. e_2 and e_3 are devices responsible for controlling a robotic



(a) Schedule without RA & Edge Resource Optimization.

(b) Schedule with RA & Edge Resource Optimization.

Figure 4.6: Example Solution Schedules.

arm p_1 in a production line. We have four critical applications for different movements, consisting of a t_i to send the movement and $t_{i.1}$ to execute it. The mapping for $t_{i.1}$ is fixed to p_1, while t_i may be executed on any device in the associated production cell, so either e_2 or e_3. $t_{i.1}$ has a data dependency on t_i, so there is a stream with a small size being sent across the network. All these applications and their tasks have a period of 1,000 µs.

Figure 4.6a shows a solution for the given scheduling problem without optimizing the resources for RA and Edge applications. Notice the long worst-case response time for an edge application, e.g., t'_1 appearing a t=0. Similarly, on the prover, notice how a large continuous time interval is taken up by critical tasks, giving an attacker ample time to perform malicious activities and possibly hide before being detected by attestation.

In comparison, Figure 4.6b shows a solution with optimizing the resources for RA and Edge applications. In this case, the worst-case response time for an edge application is significantly reduced. In addition, attestation can happen more frequently on the prover, increasing security.

4.5 Optimization Strategy

To solve the formulated problem, we use Constraint Programming (CP), which is a method to solve combinatorial optimization problems by expressing them as a set of optimization variables and constraints. CP can find the optimal solution to reasonably large realistic use cases, as we have demonstrated in our previous work [RCP21] focused on the routing and scheduling of secure TSN streams.

Our cost functions (CS1) consists of two equally weighted components: scheduling cost and extensibility cost. The scheduling cost (CS2) is equal to the sum of all critical application end-to-end latencies. The application latency is defined as the distance between the start time of the first task and the end time of the last tasks.

We use “extensibility cost” as a metric to evaluate the capability of a schedule to support RA and Edge applications. Thus, the extensibility cost in (CS3) prefers task schedules with frequent slack of even size. To achieve this, we maximize the sum of the minimum distances from each task to another; see [Bar21] for more details. Let H be the hyperperiod of the schedule, which is the least common multiple of all application periods. Let $min_dist(t)$ be a function that gives the minimum distance to another task on the same node for any task t. To transform the maximization into a minimization problem, we subtract the minimum distance from the hyperperiod.

$$cost(\Lambda^{crit}) = cost_s(\Lambda^{crit}) + cost_e(\Lambda^{crit}) \tag{CS1}$$

$$cost_s(\Lambda^{crit}) = \sum_{\lambda_i \in \Lambda^{crit}} max(\{\phi_t + t.w \mid t \in \lambda_i.\mathcal{T}\}) - min(\{\phi_t \mid t \in \lambda_i.\mathcal{T}\}) \tag{CS2}$$

$$cost_e(\Lambda^{crit}) = \sum_{\lambda_i \in \Lambda^{crit}} \sum_{t \in \lambda_i.\mathcal{T}} H - min_dist(t) \tag{CS3}$$

In the following sections, we formally define the constraints under which we minimize this cost function, whereby we reuse some of the notations and constraints from [RCP22].

4.5.1 Routing Constraints

We model the stream route variables with an integer matrix X, where the columns represent streams and rows represent nodes of the network. An entry at the position of a stream s_n and a node n in this matrix referring to a node m, represents a link from m

to n on the route of stream s_n. Alternatively, the entry could be *nil*, in which case n is not part of the route.

To determine the route for each stream $s_n \in \mathcal{S}$, for each node $n \in \mathcal{N}$ we have an optimization variable $x(s_n,n)$ representing an entry in our matrix X. The domain of $x(s_n,n)$ is defined as: $D(x(s_n,n)) = \{m \in \mathcal{N} | l_{m,n} \in \mathcal{L}\} \cup \{n\} \cup \{nil\}$. We refer to $x(s_n,n)$ as the successor of n on the path to the stream sender node. Furthermore, we use $y(s_n,n)$ to represent the length of the path from n to $s_n.t_s.e$, i.e. the length of the path from node n to the sender node of the stream. $D(y(s_n,n)) = \{i | 0 <= i <= |\mathcal{SW}| + 1\}$

We define the following constraints for the routing of streams:

$$x(s_n,n) \neq nil \Rightarrow y(s_n,n) = y(f, x(s_n,n)) + 1, \quad \forall s_n \in \mathcal{S},\ n \in \mathcal{N} \setminus \{s_n.t_s.e\} \tag{R1}$$

$$x(s_n,m) = nil \Leftrightarrow x(s_n,n) \neq m, \quad \forall s_n \in \mathcal{S},\ n,m \in \mathcal{N} \tag{R2}$$

$$x(s_n,n) \neq nil, \quad \forall s_n \in \mathcal{S},\ n \in \{t_r.e | t_r \in s_n.T_d\} \tag{R3.1}$$

$$x(s_n, s_n.t_s.e) = s_n.t_s.e, \quad \forall s_n \in \mathcal{S} \tag{R3.2}$$

$$x(s_n,n) = nil, \quad \forall s_n \in \mathcal{S},\ n \in \mathcal{E} \setminus \{t_r.e | t_r \in s_n.T_d\} \tag{R3.3}$$

$$y(s_n, s_n.t_s.e) = 0, \quad \forall s_n \in \mathcal{S} \tag{R4}$$

$$\sum_{s_d \in \mathcal{S}^d} \Big((x(s_d,n) == m \Big) \times \frac{s_d.b}{s_d.T}) \leq [m,n].s, \quad \forall n,m \in \mathcal{N} \tag{R5}$$

Please note that == and != are boolean expressions that evaluate to 1 if true and to 0 otherwise.

The constraint (R1) prevents cycles in the route, as shown in [PD12]. The constraint (R2) disallows “loose ends”, i.e., a node that has a successor/predecessor must have a predecessor/successor itself. Please note that we refer to the successor on the path from receiver to sender, i.e., the predecessor on the route. The constraint (R3.1) states that all receivers of a stream have to have a successor. Constraints (R3.2), (R3.3), and (R4) impose that the sender of the stream has itself as the successor, no other end-system has a successor, and the path length is 0 at the sender node, respectively. The constraint (R5) restricts the bandwidth usage of each link to be under 100%.

4.5.2 Task Constraints

We define the following optimization variables for tasks and streams:

- $t.e$: end-system that t is mapped to
- o^t: offset of task t (on node $t.e$)
- a^t: end-time of task t (on node $t.e$)
- o_l^s: offset of stream s on link l
- c_l^s: transmission duration of stream s on link l
- a_l^s: end-time of stream s on link l

For tasks we have the following constraints:

$$o^t + t.w = a^t, \qquad \forall t \in \mathcal{T} \tag{T1}$$

$$a^t \leq o^s_{l_{a,b}} \tag{T2.2}$$
$$\forall t \in \mathcal{T},\ s \in \mathcal{S},\ s.t_s = t$$
$$\forall l_{a,b} \in \mathcal{L} \cap \mathcal{R}^s,\ a == t.e$$

$$a^s_{l_{a,b}} \leq o^t \tag{T3}$$
$$\forall t \in \mathcal{T},\ s \in \mathcal{S},\ t \in s.T_s$$
$$\forall l_{a,b} \in \mathcal{L} \cap \mathcal{R}^s,\ b \in \mathcal{E}^s$$

$$(\alpha \times t_1.T + a^{t_1} \leq \beta \times t_2.T + o^{t_2}) \quad \vee \tag{T4}$$
$$(\beta \times t_2.T + a^{t_2} \leq \alpha \times t_1.T + o^{t_1})$$
$$\forall t_1, t_2 \in \mathcal{T},\ t_1 \neq t_2,$$
$$\forall \alpha \in \{0, ..., lcm(t_1.T, t_2.T)/t_1.T\},$$
$$\forall \beta \in \{0, ..., lcm(t_1.T, t_2.T)/t_2.T\}$$

$$\sum_{t \in \mathcal{T}} ((t.e == n) * \omega(t, n)) \leq n.C,, \tag{T5}$$
$$\forall n \in \mathcal{N}$$

The constraint (T1) sets the end-time of a task to be the sum of offset and length. The constraints (T2.2) models the dependency between a task and all its outgoing streams: such streams may only start after the task has finished. Similarly, constraint (T3) models the dependency between a task and its incoming streams: such a task may only start

after all incoming streams have arrived. Constraint (T4) prevents any two tasks from overlapping, and (T5) prevents tasks from exceeding the capacity of the node they are mapped to.

4.5.3 Stream Constraints

Finally we define the following constraints for streams:

$$cost_s(\lambda_l) \leq \lambda_l.T \quad \forall \lambda_l \in \Lambda^{crit} \tag{S1}$$

$$o_l^s = c_l^s = a_l^s = 0, \quad \forall s \in \mathcal{S},\ l_{a,b} \in \mathcal{L},\ l_{a,b} \notin \mathcal{R}^s \tag{S2}$$

$$o_l^s + c_l^s = a_l^s, \quad \forall s \in \mathcal{S},\ l_{a,b} \in \mathcal{L},\ l_{a,b} \in \mathcal{R}^s \tag{S3}$$

$$c_l^s = \left\lceil \frac{s.b}{l.s} \right\rceil, \quad \forall s \in \mathcal{S},\ l_{a,b} \in \mathcal{L},\ l_{a,b} \in \mathcal{R}^s \tag{S4}$$

$$a_{l_{a,b}}^s \leq o_{l_{b,c}}^s \quad \forall s \in \mathcal{S},\ l_{b,c} \in \mathcal{L} \cap \mathcal{R}^s, a = x(s,b) \tag{S5}$$

The constraint (S1) sets the deadline for the completion of an application to its period. The constraints (S2) sets all optimization variables to zero for all links not part of a streams route. For all other links and nodes (S3) sets the end-time to be the sum of offset a length. For each link on the route of a stream constraint, (S4) sets the length to be the byte-size of the stream divided by the link-speed. Constraint (S5) enforces that a stream is scheduled consecutively along its route.

$$\begin{aligned}
&(\alpha \times s_1.T + a_l^{s_1} \leq \beta \times s_2.T + o_l^{s_2}) \quad \vee \\
&(\beta \times s_2.T + a_l^{s_2} <= \alpha \times s_1.T + o_l^{s_1}) \\
&\quad \forall s_1, s_2 \in \mathcal{S}, s_1 \neq s_2, \forall l \in \mathcal{R}_1^s \cap \mathcal{R}_2^s, \\
&\quad \forall \alpha \in \{0, ..., lcm(s_1.T, s_2.T)/s_1.T\}, \\
&\quad \forall \beta \in \{0, ..., lcm(s_1.T, s_2.T)/s_2.T\}
\end{aligned} \tag{S6}$$

$$(\alpha \times s_2.T + o^{s_2}_{l_{b,c}} <= \beta \times s_1.T + o^{s_1}_{l_{a_1,b}}) \quad \vee \tag{S7}$$
$$(\beta \times s_1.T + o^{s_1}_{l_{b,c}} <= \alpha \times s_2.T + o^{s_2}_{l_{a_2,b}})$$
$$\forall s_1, s_2 \in \mathcal{S}, s_1 \neq s_2, \forall l \in \mathcal{R}^s_1 \cap \mathcal{R}^s_2,$$
$$a_1 = x(s_1, b),\ a_2 = x(s_2, b),$$
$$\forall \alpha \in \{0, ..., lcm(s_1.T, s_2.T)/s_1.T\},$$
$$\forall \beta \in \{0, ..., lcm(s_1.T, s_2.T)/s_2.T\}$$

The constraint (S6) prevents any streams from overlapping on any nodes or links. Furthermore, constraint (S7) guarantees that for each link connected to an output port of a switch, the frames arriving on all input ports of that switch that want to use this output port cannot overlap in the time domain. This is the frame isolation necessary for determinism in our TSN configuration, which is further explained in [CSCS16].

4.6 Experimental Evaluation

For the evaluation, we are interested in measuring the impact of our extensiblity- and RA-aware solution on the formulated problem.

To this extent, we created an industrial-inspired use case, inspired by [Bar21]. The architecture can be seen in Figure 4.1. The nodes e_1, v_1 and v_2 are high-powered edge servers running on-premise in a secure room in a factory. v_1 and v_2 were chosen as verifiers as they are the most trusted and protected machines. e_1 is connected to the cloud and responsible for data analytics. The factory consists of two production floors (cells). On the first floor there are two control systems e_2 and e_3 that control a robotic arm p_1 and a conveyor belt p_2. On the second floor there is a control system e_4 that controls an industrial oven p_3 and gets readings from a temperature sensor p_4. The systems p_1 to p_4 were identified as the most safety-critical and should thus be regularly verified. v_1 will do this for p_1 and p_2, while v_2 will do it for p_3 and p_4. Attestation of e_2, e_3 and e_4 could also make sense, but is out of scope for this work because higher-powered systems can use different RA techniques.

Overall, the use case consists of 19 critical applications, 27 critical tasks and 8 streams. Furthermore, we created 7 sets, labelled E1-E7, of 12 non-critical edge applications, each with 1–2 tasks, half of which exchange non-critical streams. The edge applications implement data analytics and diagnostics and have random arrival times. We ran two experiments: **NOEXT** without the extensibility cost (see Section 4.5) and **EXT** with this cost. Our solution was implemented in Python 3.9 using the CP-SAT solver from Google OR-Tools [Goo]. It was run on a machine with an i7-8565U CPU with

	NOEXT	**EXT**
T^r_{max}	425	150
T^e_{wc}	350	233
T^e_{avg}	66421	45117
L	4668	5141

Table 4.1: Evaluation Results.

	NOEXT	**EXT**
E1	121.75	79.75 (-34.5%)
E2	180.83	107.83 (-40.37%)
E3	215.0	185.17 (-13.87%)
E4	166.25	148.33 (-10.78%)
E5	161.67	134.67 (-16.7%)
E6	109.83	109.0 (-0.76%)
E7	101.5	65.33 (-35.64%)

Table 4.2: Impact of Extensibility Formulation on Average Latency of Edge Applications.

16 GB of DDR4-RAM. [1]

The results can be seen in Table 4.1. T^r_{max} is the average maximum unattested time among all provers. The longer a prover stays unattested, the more time an attacker has to execute malicious code and hide itself before the next attestation. T^e_{wc} is the average worst-case response time for edge applications, and T^e_{avg} is the average average-case response time. These averages are taken across all nodes that could execute edge applications (all nodes except provers) and assume the edge application to appear at a random time. L is the sum of critical application latencies, i.e., $cost_s$ from Equation CS2.

As the results show, in **EXT** the average unattested time is reduced by 64.7%, resulting in better security. Additionally, in **EXT** the average and worst-case response times for dynamic edge application are significantly improved. These improvements come at a slight cost in the form of increased latency for critical applications. However, these applications can still easily meet their deadlines.

The improvement in response time for edge applications is also shown by our experiments with the randomly generated set of edge applications E1-E7 in Table 4.2. For each of these sets we measured the average end-to-end latency both for NOEXT and EXT. On average, edge applications have 21% smaller latency with our solution.

[1] The tool including the obtained results is available on GitHub: `https://github.com/nreusch/TSNConf`

4.7 Conclusions and Future Work

Edge Computing is an enabler for Industry 4.0 where mixed-criticality applications are running on a shared computing platform, which guarantees their safety, security and performance. In this paper, we considered an ECP that uses TSN for the communication and RA for security, and implements mixed-criticality applications. The critical applications are scheduled with static cyclic scheduling, whereas the noncritical Edge applications are handled dynamically at runtime.

We proposed a CP-based solution to the problem of mapping and scheduling the mixed-criticality applications on an ECP, which decides the mapping of tasks to devices and their schedule tables, as well as the GCLs for the TSN communication. As the use case has shown, our approach is able to guarantee the deadlines of the critical applications at design time, provision resources to perform RA, and successfully accommodate dynamic responsive Edge applications at runtime with a shorter response time.

In our future work, we want to do a more extensive evaluation and propose a metaheuristic-based solution to handle large realistic use cases.

Appendix A

TSNConf: Testcase and Schedule Visualization Tool

The methods presented in this thesis have been implemented as open-source software prototypes, available for download from public repositories. We collectively call these software prototypes “TSNConf” from “TSN configuration tools”. In this appendix, we present the visualization component of TSNConf, used to visualize our test cases and help us run our experiments. This has proven helpful, e.g., when developing heuristics, to visualize example test cases and have a graphic representation of the topology, routing, and schedule, instead of imagining and remembering all these things. When using Constraint Programming, it was useful to quickly see the changes to the resulting schedule under certain constraints, without having to parse a text file.

The tool was developed using Python 3[1] and uses the Dash framework[2] for a reactive browser interface. It is openly available on GitHub[3] and can be tested in the browser[4]. The tool has the following main features:

- Visualization of network architecture and task mapping using GraphViz[5] plots

[1] https://www.python.org/
[2] https://dash.plotly.com/introduction
[3] https://github.com/nreusch/TSNConf
[4] https://tsnconf-demo.herokuapp.com/
[5] https://graphviz.org/

- Visualization of application DAGs using GraphViz[5] plots
- Structured presentation of task & message information
- Interactive visualization of message routes
- Interactive visualization of task & message schedule

The tool is executed on the command line using a Python interpreter, requiring a *network description file* as input. The network description file uses an easy-to-read XML syntax to describe a test case, as shown in Listing A.1. It can also translate to and from various other formats, including the format necessary for OMNet++ simulations with NeSTiNg.

Figure A.1 shows the visualization of an example network architecture, including the mapping of tasks to different end-systems. The graph is described in the DOT[6] language and visualized using a GraphViz library for Python. It is also exported as DOT- and PDF-file.

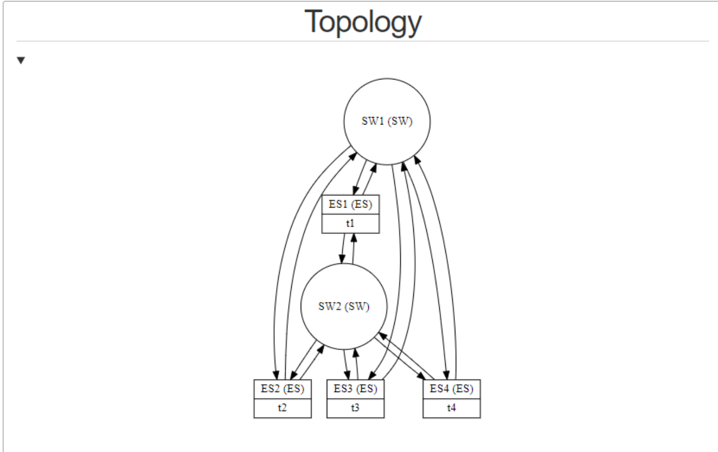
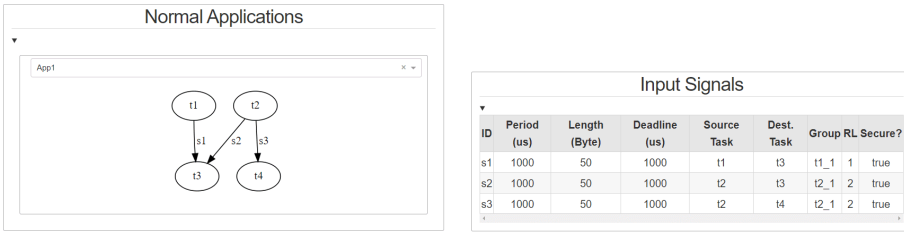


Figure A.1: TSNConf: Network Architecture Visualization.

DAGs representing applications, i.e., tasks with message dependencies, are also described in DOT language and rendered using GraphViz, see for example Figure A.2a. The widget includes a search bar. Other available information about tasks and messages is displayed in various tables, see, for example, Figure A.2b.

[6]https://graphviz.org/doc/info/lang.html



ID	Period (us)	Length (Byte)	Deadline (us)	Source Task	Dest. Task	Group	RL	Secure?
s1	1000	50	1000	t1	t3	t1_1	1	true
s2	1000	50	1000	t2	t3	t2_1	2	true
s3	1000	50	1000	t2	t4	t2_1	2	true

(a) Application Visualization **(b)** Information Tables

Figure A.2: TSNConf: Various Visualizations.

Routing of messages is visualized as a Cytoscape[7] network. Individual routes can be toggled on and off. The graph can be arranged according to different layout algorithms (Cose-Bilkent, Euler, Dagre, etc.) or by dragging nodes around. See Figure A.3 for an example.

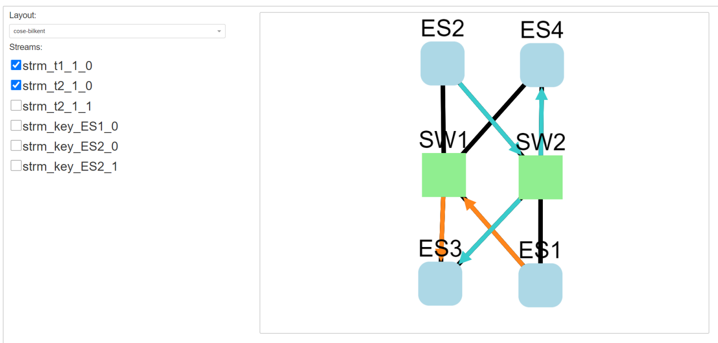


Figure A.3: TSNConf: Routing visualization.

The resulting combined network- and task-schedule is visualized as a customized bar chart. Different categories of tasks and message can be displayed in different colors and can enabled/disabled individually. Hovering over a block gives detailed information about start time, end time etc. It is possible to zoom into selected areas and export the schedule as PDF, SVG or interactive HTML. See Figure A.4 for an example.

The tool uses a modular architecture and a flexible XML-parser for the input test cases. This allows a user to work with an input model of his own liking and omit or extend parts of our own models. This way, the tool can be used to help solve a wide range of problems. It uses Python's pickling feature to output files, from which all the visualizations and results can be restored without re-running time intensive optimization.

[7] https://dash.plotly.com/cytoscape

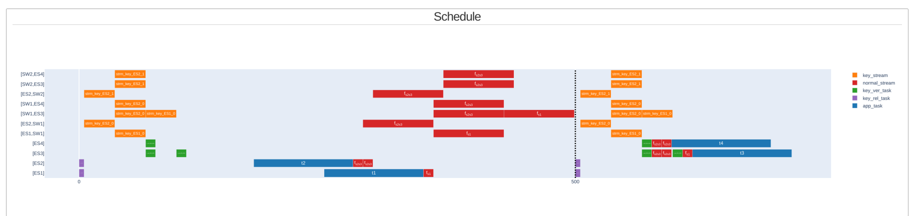


Figure A.4: TSNConf: Schedule Visualization

```xml
<NetworkDescription mtu="1500" frame_overhead="16" key_length="16" mac_length="22">
	<device name="SW1" type="Switch"/>
	<device name="SW2" type="Switch"/>
	<device name="ES1" type="EndSystem" mac_exec_time="10"/>
	...

	<link src="SW1" dest="ES1" speed="12.50"/>
	...
	<link src="ES4" dest="SW2" speed="12.50"/>

	<application name="App1" period="1000" type="NORMAL">
		<tasks>
			<task name="t1" node="ES1" wcet="100" period="1000" arrival_time="
			     0" type="NORMAL"/>
			...
		</tasks>
		<streams>
			<stream name="s1" src="ES1" dest="ES3" sender_task="t1"
			     receiver_tasks="t3" size="538" period="1000" rl="1" secure="
			     True" type="NORMAL" />
			...
		</streams>
	</application>
	...



	<route stream="s1_0">
		<link src="ES1" dest="SW1" />
		<link src="SW1" dest="ES3" />
	</route>
	...

	<schedule>
		<node src="ES1" dest="ES1">
			<block start="258" duration="100" end="358" creator="t1"/>
			...
		</node>
		...
		<costs>
			<cost app_id="App1" value="405"/>
			<cost app_id="SecApp_ES1" value="23"/>
			<cost app_id="SecApp_ES2" value="23"/>
		</costs>
	</schedule>
</NetworkDescription>
```

Listing A.1: TSNConf: Network Description File

Bibliography

[ADD21] Sigurd Frej Joel Jørgensen Ankergård, Edlira Dushku, and Nicola Dragoni. State-of-the-art software-based remote attestation: Opportunities and open issues for internet of things. *Sensors*, 21(5):1598, 2021.

[AEP18] Amir Aminifar, Petru Eles, and Zebo Peng. Optimization of message encryption for real-time applications in embedded systems. *IEEE Transactions on Computers*, 67(5):748–754, 2018.

[AHM20] Ayman A. Atallah, Ghaith Bany Hamad, and Otmane Ait Mohamed. Routing and scheduling of time-triggered traffic in time-sensitive networks. *IEEE Transactions on Industrial Informatics*, 16(7):4525–4534, 2020.

[AJP+22] Mette Abildgaard, Leif Lahn Jensen, Troels Lund Poulsen, Sophie Løhde, Mikkel Irminger Sarbo, and Serdal Benli. Statens it-beredskab. Technical report, Statsrevisorerne, 2022.

[ALD+21] Mohammad Ashjaei, Lucia Lo Bello, Masoud Daneshtalab, Gaetano Patti, Sergio Saponara, and Saad Mubeen. Time-sensitive networking in automotive embedded systems: State of the art and research opportunities. *JSA*, 117:102137, 2021.

[BAH+22] Sushmit Bhattacharjee, Konstantinos Alexandris, Emil Hansen, Paul Pop, and Thomas Bauschert. Latency-aware function placement, routing, and scheduling in tsn-based industrial networks. In *ICC 2022 - IEEE International Conference on Communications*, pages 4248–4254, 2022.

[Bar21] Mohammadreza Barzegaran. *Configuration Optimization of Fog Computing Platforms for Control Applications*. PhD thesis, Technical University of Denmark, 2021.

[BCP20] Mohammadreza Barzegaran, Anton Cervin, and Paul Pop. Performance optimization of control applications on fog computing platforms using scheduling and isolation. *IEEE Access*, 8:104085–104098, 2020.

[BD17] Alan Burns and Robert I. Davis. A survey of research into mixed criticality systems. *ACM Comput. Surv.*, 50(6), nov 2017.

[BDF01] Dan Boneh, Glenn Durfee, and Matt Franklin. Lower bounds for multicast message authentication. *Advances in Cryptology — Eurocrypt 2001*, pages 437–452, 2001.

[BDNM16] M. Boyer, H. Daigmorte, N. Navet, and J. Migge. Performance impact of the interactions between time-triggered and rate-constrained transmissions in TTEthernet. In *Proc. ERTS*, 2016.

[BG11] Radhakisan Baheti and Helen Gill. Cyber-physical systems. *The impact of control technology*, 12(1):161–166, 2011.

[BK+05] E. K. Burke, G. Kendall, et al. *Search methodologies*. Springer, New York, NY, 2005.

[BP22] Mohammadreza Barzegaran and Paul Pop. Extensibility-aware fog computing platform configuration for mixed-criticality applications. *Journal of Systems Architecture*, 133:102776, 2022.

[BRST16] Ferdinand Brasser, Kasper B. Rasmussen, Ahmad Reza Sadeghi, and Gene Tsudik. Remote attestation for low-end embedded devices: The prover's perspective. *Proceedings of the Design Automation Conference*, page a91, 2016.

[BRZ+22] Mohammadreza Barzegaran, Niklas Reusch, Luxi Zhao, Silviu S. Craciunas, and Paul Pop. Real-time traffic guarantees in heterogeneous time-sensitive networks. In *Proc. RTNS*, page 46–57, New York, NY, USA, 2022. ACM.

[But11] Giorgio C. Buttazzo. *Hard real-time computing systems: Predictable scheduling algorithms and applications*. Springer, 2011.

[BW21] Martin Böhm and Diederich Wermser. Multi-domain time-sensitive networks—control plane mechanisms for dynamic inter-domain stream configuration. *Electronics*, 10(20), 2021.

[BZP20] Mohammadreza Barzegaran, Bahram Zarrin, and Paul Pop. Quality-Of-Control-Aware Scheduling of Communication in TSN-Based Fog Computing Platforms Using Constraint Programming. In *Proc. Fog-IoT*. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, Germany, 2020.

[CAN11] Thomas M. Chen and Saeed Abu-Nimeh. Lessons from stuxnet. *Computer*, 44(4):5742014, 2011.

[CMP+10] Daniel Cordeiro, Grégory Mounié, Swann Perarnau, Denis Trystram, Jean-Marc Vincent, and Frédéric Wagner. Random graph generation for scheduling simulations. In *Proc. SIMUTools*. ICST, 2010.

[Coma] International Electrotechnical Commission. IEC 61784-2-12, Industrial communication networks-Profiles, Part 2: CPF 12 EtherCAT.

[Comb] International Electrotechnical Commission. IEC 61784-2-3, Industrial communication networks-Profiles, Part 2: CPF 3 PROFIBUS & PROFINET.

[CRT18] Xavier Carpent, Norrathep Rattanavipanon, and Gene Tsudik. Remote attestation of iot devices via smarm: Shuffled measurements against roving malware. In *IEEE International Symposium on Hardware Oriented Security and Trust*, pages 9–16, 2018.

[CSCS16] Silviu S. Craciunas, Ramon Serna Oliver, Martin Chmelík, and Wilfried Steiner. Scheduling Real-Time Communication in IEEE 802.1Qbv Time Sensitive Networks. In *Proceedings of International Conference on Real-Time Networks and Systems (RTNS)*, pages 183–192, 2016.

[CSO16] Silviu S. Craciunas and Ramon Serna Oliver. Combined task- and network-level scheduling for distributed time-triggered systems. *Journal of Real-Time Systems*, 52(2):161–200, 2016.

[CSO17] S. S. Craciunas and R. Serna Oliver. An overview of scheduling mechanisms for time-sensitive networks. Technical report, Real-time summer school L'École d'Été Temps Réel (ETR), 2017.

[CTHC15] Shiwei Cao, Xiaofeng Tao, Yanzhao Hou, and Qimei Cui. An energy-optimal offloading algorithm of mobile computing based on hetnets. In *2015 International Conference on Connected Vehicles and Expo (ICCVE)*, pages 254–258, 2015.

[CWW+16] William Arthur Conklin, Gregory B. White, Dwayne. Williams, Roger Davis, and Chuck. Cothren. *Principles of computer security*. McGraw-Hill Education, 2016.

[DAB14] J. A. R. De Azua and M. Boyer. Complete modelling of AVB in network calculus framework. In *Proc. RTNS*, pages 55–64, 2014.

[Dec05] J-D Decotignie. Ethernet-based real-time and industrial communications. *Proceedings of the IEEE*, 93(6):1102–1117, 2005.

[DN16] F. Dürr and 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*, pages 203–212, 2016.

[DTE12] J. Diemer, D. Thiele, and R. Ernst. Formal worst-case timing analysis of ethernet topologies with strict-priority and AVB switching. In *Proc. SIES*, pages 1–10, 2012.

[DTF16] Maofei Deng, Hui Tian, and Bo Fan. Fine-granularity based application offloading policy in cloud-enhanced small cell networks. *2016 Ieee International Conference on Communications Workshops, Icc 2016*, pages 638–643, 2016.

[DWZ21] Jialin Dai, Zhongcheng Wang, and Long Zhong. Research on gating scheduling of time sensitive network based on constraint strategy. *Journal of Physics: Conference Series*, 1920(1):012089, 2021.

[DXL+23] Libing Deng, Guoqi Xie, Hong Liu, Yunbo Han, Renfa Li, and Keqin Li. A survey of real-time ethernet modeling and design methodologies: From avb to tsn. *Acm Computing Surveys*, 55(2):3487330, 2023.

[EBN+21] Doganalp Ergenc, Cornelia Brulhart, Jens Neumann, Leo Kruger, and Mathias Fischer. On the security of ieee 802.1 time-sensitive networking. *2021 Ieee International Conference on Communications Workshops, Icc Workshops 2021 - Proceedings*, page 9473542, 2021.

[EDPT12] Karim El Defrawy, Daniele Perito, and Gene Tsudik. Smart: Secure and minimal architecture for (establishing dynamic) root of trust. In *Proceedings of Network and Distributed System Security Symposium*, volume 12, pages 1–15, 2012.

[ESA96] Ariane 5 - flight 501 failure. Technical report, European Space Agency, 1996.

[FCD22] Zhiwei Feng, Mingyang Cai, and Qingxu Deng. An efficient proactive fault-tolerance scheduling of ieee 802.1qbv time-sensitive network. *Ieee Internet of Things Journal*, 9(16):14501–14510, 2022.

[FDCL22] Zhiwei Feng, Qingxu Deng, Mingyang Cai, and Jinghua Li. Efficient reservation-based fault-tolerant scheduling for ieee 802.1qbv time-sensitive networking. *Journal of Systems Architecture*, 123:102381, 2022.

[FDR18] Jonathan Falk, Frank Dürr, and Kurt Rothermel. Exploring practical limitations of joint routing and scheduling for TSN with ILP. In *Proc. RTCSA*, 2018.

[FHC+19] J. Falk, D. Hellmanns, B. Carabelli, N. Nayak, F. Dürr, S. Kehrer, and K. Rothermel. NeSTiNg: Simulating IEEE time-sensitive networking (TSN) in OMNeT++. In *Proc. NetSys*, pages 1–8, 2019.

[FY21] Tao Feng and Hao Yang. Smt-based task- and network-level static schedule for time sensitive network. *2021 Ieee 3rd International Conference on Communications, Information System and Computer Engineering, Cisce 2021*, pages 764–770, 2021.

[GBI21] Manjari Gupta, Lava Bhargava, and S. Indu. Mapping techniques in multicore processors: current and future trends. *Journal of Supercomputing*, 77(8):9308–9363, 2021.

[GHKS98] Miltos D Grammatikakis, D.Frank Hsu, Miro Kraetzl, and Jop F Sibeyn. Packet routing in fixed-connection networks: A survey. *Journal of Parallel and Distributed Computing*, 54(2):77–132, 1998.

[Goo] Google Inc. *CP-SAT Solver Guide*.

[Goo20] Google. Google OR-Tools. `https://developers.google.com/optimization`, Accessed on Oct 2020.

[GP18] Voica Gavriluţ and Paul Pop. Scheduling in time sensitive networks (tsn) for mixed-criticality industrial applications. In *2018 14th IEEE International Workshop on Factory Communication Systems (WFCS)*, pages 1–4, 2018.

[GP20] Voica Gavriluţ and Paul Pop. Traffic-type assignment for tsn-based mixed-criticality cyber-physical systems. *ACM Trans. Cyber-Phys. Syst.*, 4(2), 2020.

[GT14] Michael Goodrich and Roberto Tamassia. *Introduction to computer security*. Pearson Education,, 2014.

[GZPS17] Voica Gavriluţ, Bahram Zarrin, Paul Pop, and Soheil Samii. Fault-tolerant topology and routing synthesis for IEEE time-sensitive networking. In *Proc. RTNS*. ACM, 2017.

[GZRP18] Voica Gavriluţ, Luxi Zhao, Michael L. Raagaard, and Paul Pop. AVB-aware routing and scheduling of time-triggered traffic for TSN. *IEEE Access*, 6:75229–75243, 2018.

[HAD+21] Bahar Houtan, Mohammad Ashjaei, Masoud Daneshtalab, Mikael Sjödin, and Saad Mubeen. Synthesising schedules to improve qos of best-effort traffic in TSN networks. In *Proc. RTNS*. ACM, 2021.

[Hau22] Nana Haugaard. Sjællandsk forsyning hacket og afkrævet løsesum. *Version2*, 2022.

[HF19] F. Heilmann and G. Fohler. Size-based queuing: An approach to improve bandwidth utilization in TSN networks. *SIGBED Rev.*, 16(1):9–14, 2019.

[HFG+20] D. Hellmanns, J. Falk, A. Glavackij, R. Hummen, S. Kehrer, and F. Dürr. On the performance of stream-based, class-based time-aware shaping and frame preemption in TSN. In *Proc. ICIT*, pages 298–303, 2020.

[HGF+20] D. Hellmanns, A. Glavackij, J. Falk, F. Duerr, R. Hummen, and S. Kehrer. Scaling TSN scheduling for factory automation networks. In *Proc. WFCS*, pages 1–8, 2020.

[HHLBS09] Frank Hutter, Holger H. Hoos, Kevin Leyton-Brown, and Thomas Stützle. ParamILS: an automatic algorithm configuration framework. *Journal of Artificial Intelligence Research*, 36:267–306, October 2009.

[HHSH06] Carl Hartung, Richard Han, Carl Seielstad, and Saxon Holbrook. Firewxnet: A multi-tiered portable wireless system for monitoring weather conditions in wildland fire environments. *Mobisys 2006 - Fourth International Conference on Mobile Systems, Applications and Services*, 2006:28–41, 2006.

[HSS08] Aric A. Hagberg, Daniel A. Schult, and Pieter J. Swart. Exploring network structure, dynamics, and function using networkx. In *Proc. SciPy*, 2008.

[HWW+21] Kai Huang, Xinming Wan, Ke Wang, Xiaowen Jiang, Junjian Chen, Qingtang Deng, Wenyuan Xu, Yonggang Peng, and Zhili Liu. Reliability-aware multipath routing of time-triggered traffic in time-sensitive networks. *Electronics*, 10(2), 2021.

[Ins11] Institute of Electrical and Electronics Engineers, Inc. 802.1BA—Audio Video Bridging (AVB) Systems. `http://www.ieee802.org/1/pages/802.1ba.html`, 2011. Accessed: 31.12.2022.

[Ins14] Institute of Electrical and Electronics Engineers, Inc. 802.1Q-2014 - Bridges and Bridged Networks. http://www.ieee802.org/1/pages/802.1Q.html, 2014. Accessed: 31.12.2022.

[Ins16a] Institute of Electrical and Electronics Engineers, Inc. 802.1Qbu - Frame Preemption. http://www.ieee802.org/1/pages/802.1bu.html, 2016. Accessed: 31.12.2022.

[Ins16b] Institute of Electrical and Electronics Engineers, Inc. 802.1Qbv - Enhancements for Scheduled Traffic. http://www.ieee802.org/1/pages/802.1bv.html, 2016. Draft 3.1, Accessed: 31.12.2022.

[Ins16c] Institute of Electrical and Electronics Engineers, Inc. Official Website of the 802.1 Time-Sensitive Networking Task Group. http://www.ieee802.org/1/pages/tsn.html, 2016. Accessed: 31.12.2022.

[Ins17a] Institute of Electrical and Electronics Engineers, Inc. 802.1AS-Rev - Timing and Synchronization for Time-Sensitive Applications. http://www.ieee802.org/1/pages/802.1AS-rev.html, 2017. Accessed: 31.12.2022.

[Ins17b] Institute of Electrical and Electronics Engineers, Inc. 802.1CB - Frame Replication and Elimination for Reliability. https://standards.ieee.org/standard/802_1CB-2017.html, 2017. Accessed: 31.12.2022.

[Ins17c] Institute of Electrical and Electronics Engineers, Inc. 802.1Qci - Per-Stream Filtering and Policing. https://standards.ieee.org/standard/802_1Qci-2017.html, 2017. Accessed: 31.12.2022.

[Int10] International Electrotechnical Commission. IEC 61508: Functional safety of electrical/electronic/programmable electronic safety-related systems, 2010.

[IPEP05] V. Izosimov, P. Pop, P. Eles, and Z. Peng. Design optimization of time- and cost-constrained fault-tolerant distributed embedded systems. In *Proc. DATE*, 2005.

[Iss11] Issuing Committee: As-2d2 Deterministic Ethernet And Unified Networking. SAE AS6802 Time-Triggered Ethernet. http://standards.sae.org/as6802/, 2011. Accessed: 31.12.2022.

[JEP12] Ke Jiang, Petru Eles, and Zebo Peng. Co-design techniques for distributed real-time embedded systems with communication security constraints. In *2012 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 947–952, 2012.

[JPJ17] Wei Jiang, Paul Pop, and Ke Jiang. Design optimization for security- and safety-critical distributed real-time applications. *Microprocessors and Microsystems*, 52:401–415, 2017.

[JYP01] Le Boudec Jean-Yves and Thiran Patrick. Network calculus: a theory of deterministic queuing systems for the internet. *Real-Time Syst.*, 51, 2001.

[KGJV83] S. Kirkpatrick, C. D. Gelatt, Jr., and M. P. Vecchi. Optimization by simulated annealing. *Science*, 220:671–680, 1983.

[KHM05] Nagarajan Kandasamy, John P. Hayes, and Brian T. Murray. Dependable communication synthesis for distributed embedded systems. *Reliability Engineering and System Safety*, 89(1):81–92, 2005.

[KK12] Kyoung-Dae Kim and P. R. Kumar. Cyber–physical systems: A perspective at the centennial. *Proceedings of the IEEE*, 100(Special Centennial Issue):1287–1308, 2012.

[KLC$^+$10] Jeonggil Ko, Jong Hyun Lim, Yin Chen, Răzvan Musăloiu-E, Andreas Terzis, Gerald M. Masson, Tia Gao, Walt Destler, Leo Selavo, and Richard P. Dutton. Medisn: Medical emergency detection in sensor networks. *Transactions on Embedded Computing Systems*, 10(1):11, 2010.

[KM14] D. D. Khanh and A. Mifdaoui. Timing Analysis of TDMA-based Networks using Network Calculus and Integer Linear Programming. In *Proc. MASCOTS*, pages 21–30, 2014.

[Kni02] John C. Knight. Safety critical systems: Challenges and directions. In *Proceedings of the 24th International Conference on Software Engineering*, ICSE '02, page 547–550, New York, NY, USA, 2002. Association for Computing Machinery.

[KP17] Vasileios Karagiannis and Apostolos Papageorgiou. Network-integrated edge computing orchestrator for application placement. In *IEEE International Conference on Network and Service Management*, pages 1–5, 2017.

[KS22] Hermann Kopetz and Wilfried Steiner. *Real-Time Systems : Design Principles for Distributed Embedded Applications, Real-Time Systems*. Springer International Publishing AG, 2022.

[KTR$^+$21] Eleftherios Kyriakakis, Koen Tange, Niklas Reusch, Eder Ollora Zaballa, Xenofon Fafoutis, Martin Schoeberl, and Nicola Dragoni. Fault-tolerant clock synchronization using precise time protocol multi-domain aggregation. *Proceedings of 2021 Ieee 24th*

International Symposium on Real-time Distributed Computing, pages 114–122, 2021.

[KZH15] Simon Kramer, Dirk Ziegenbein, and Arne Hamann. Real world automotive benchmarks for free. In *Proc. WATERS*, 2015.

[LBS19] Lucia Lo Bello and Wilfried Steiner. A perspective on ieee time-sensitive networking for industrial communication and automation systems. *Proceedings of the IEEE*, 107(6):1094–1120, 2019.

[Lee17] R Lee. CRASHOVERRIDE: Analysis of the threat to electric grid operations. Technical report, Dragos Inc., 2017.

[LFK+14] Heiner Lasi, Peter Fettke, Hans-Georg Kemper, Thomas Feld, and Michael Hoffmann. Industry 4.0. *Business & Information Systems Engineering*, 6(4):239–242, 2014.

[LLEM+20] Ana Larrañaga, M. Carmen Lucas-Estañ, Imanol Martinez, Iñaki Val, and Javier Gozalvez. Analysis of 5G-TSN integration to support industry 4.0. In *Proc. ETFA*, 2020.

[LPS16] Sune Mølgaard Laursen, Paul Pop, and Wilfried Steiner. Routing optimization of AVB streams in TSN networks. *SIGBED Rev.*, 13(4):43–48, November 2016.

[MAP+21] Daniel Bujosa Mateu, Mohammad Ashjaei, Alessandro V. Papadopoulos, Julian Proenza, and Thomas Nolte. LETRA: mapping legacy ethernet-based traffic into TSN traffic classes. In *Proc. ETFA*, 2021.

[MAS+18] Rouhollah Mahfouzi, Amir Aminifar, Soheil Samii, Ahmed Rezine, Petru Eles, and Zebo Peng. Stability-aware integrated routing and scheduling for control applications in ethernet networks. *Proceedings of the 2018 Design, Automation and Test in Europe Conference and Exhibition, Date 2018*, 2018-:682–687, 2018.

[MAS+19] Rouhollah Mahfouzi, Amir Aminifar, Soheil Samii, Petru Eles, and Zebo Peng. Security-aware routing and scheduling for control applications on ethernet TSN networks. *ACM Trans. Des. Autom. Electron. Syst.*, 25(1), 2019.

[MHKS19] Philipp Meyer, Timo Häckel, Franz Korf, and Thomas C. Schmidt. Dos protection through credit based metering – simulation-based evaluation for time-sensitive networking in cars. 2019.

[MJHPC22] Shane D. McLean, Emil Alexander Juul Hansen, Paul Pop, and Silviu S. Craciunas. Configuring adas platforms for automotive applications using metaheuristics. *Frontiers in Robotics and Ai*, 8:762227, 2022.

[MVOV96] Alfred J. Menezes, Paul C. Van Oorschot, and Scott A. Vanstone. *Handbook of applied cryptography*. CRC Press, 1996.

[MYPB14] Sibin Mohan, Man Ki Yoon, Rodolfo Pellizzoni, and Rakesh Bobba. Real-time systems security through scheduler constraints. *Proceedings - Euromicro Conference on Real-time Systems*, pages 129–140, 2014.

[NDR18a] N. G. Nayak, F. Dürr, and K. Rothermel. Incremental flow scheduling and routing in time-sensitive software-defined networks. *IEEE Trans Industr Inform*, 14(5), 2018.

[NDR18b] Naresh Ganesh Nayak, Frank Dürr, and Kurt Rothermel. Routing algorithms for IEEE 802.1Qbv networks. *SIGBED Rev.*, 15(3):13–18, 2018.

[NEL90] VP NELSON. Fault-tolerant computing - fundamental-concepts. *Computer*, 23(7):19–25, 1990.

[NMFMMT22] Allan Nisgaard, Marcel Mirzaei-Fard, Henrik Moltke, and Ingeborg Munk Toft. Var tæt på at slukke tusindvis af vindmøller: Nu fortæller vestas om cyberangreb. *Danish Broadcasting Corporation*, 2022.

[OY20] Mubarak Adetunji Ojewale and Patrick Meumeu Yomsi. Routing heuristics for load-balanced transmission in TSN-based networks. *SIGBED Rev.*, 16(4):20–25, January 2020.

[PBA17] T. Pereira, L. Barreto, and A. Amaral. Network and information security challenges within industry 4.0 paradigm. *Procedia Manufacturing*, 13:1253–1260, 2017.

[PCST01] Adrian Perrig, Ran Canetti, Dawn Song, and J Doug Tygar. Efficient and secure source authentication for multicast. In *Proc. NDSS*, volume 1, 2001.

[PCTS02] Adrian Perrig, Ran Canetti, J. D. Tygar, and Dawn Song. The TESLA broadcast authentication protocol. *RSA CRYPTOBYTES*, page 2002, 2002.

[PD12] Quang Dung Pham and Yves Deville. Solving the quorumcast routing problem by constraint programming. *Constraints*, 17(4):409–431, 2012.

[PLCS16] P. Pop, M. L. Raagaard, S. S. Craciunas, and W. Steiner. Design optimization of cyber-physical distributed systems using IEEE Time-Sensitive networks (TSN). *IET Cyber-Physical Systems: Theory and Applications*, 1(1):86–94, 2016.

[PML+19] Carlo Puliafito, Enzo Mingozzi, Francesco Longo, Antonio Puliafito, and Omer Rana. Fog computing for the internet of things: A survey. *ACM Transactions on Internet Technology*, 19(2):1–41, 2019.

[PO18] Maryam Pahlevan and Roman Obermaisser. Genetic algorithm for scheduling time-triggered traffic in time-sensitive networks. In *Proc. ETFA*, 2018.

[PPEP06] Traian Pop, Paul Pop, Petru Eles, and Zebo Peng. Timing analysis of the flexray communication protocol. *Euromicro Conference on Real-time Systems*, 2006:203–213, 2006.

[Pry08] G. Prytz. A performance analysis of EtherCAT and PROFINET IRT. In *Proc. ETFA*. IEEE Computer Society, 2008.

[PSC+05] A. Perrig, D. Song, R. Canetti, J. D. Tygar, and B. Briscoe. Timed efficient stream loss-tolerant authentication (TESLA): Multicast source authentication transform introduction. RFC 4082, June 2005.

[PSS19] Taeju Park, Soheil Samii, and Kang G. Shin. Design optimization of frame preemption in real-time switched ethernet. *Proceedings of the 2019 Design, Automation and Test in Europe Conference and Exhibition, Date 2019*, pages 420–425, 2019.

[PTO19] Maryam Pahlevan, Nadra Tabassam, and Roman Obermaisser. Heuristic list scheduler for time triggered traffic in time sensitive networks. *SIGBED Rev.*, 16(1):15–20, 2019.

[PZB+21] Paul Pop, Bahram Zarrin, Mohammadreza Barzegaran, Stefan Schulte, Sasikumar Punnekkat, Jan Ruh, and Wilfried Steiner. The fora fog computing platform for industrial iot, 2021.

[Rau14] Marvin Rausand. *Reliability of Safety-Critical Systems: Theory and Applications*, volume 9781118112724. Wiley Blackwell, 2014.

[RCP21] Niklas Reusch, Silviu S. Craciunas, and Paul Pop. Dependability-aware routing and scheduling for time-sensitive networking, 2021.

[RCP22] Niklas Reusch, Silviu S. Craciunas, and Paul Pop. Dependability-aware routing and scheduling for time-sensitive networking. *Iet Cyber-physical Systems: Theory and Applications*, 7(3):124–146, 2022.

[RP17] M. L. Raagaard and P. Pop. Optimization algorithms for the scheduling of IEEE 802.1 Time-Sensitive Networking (TSN). Technical report, 2017.

[RP21] Niklas Reusch and Paul Pop. Scheduling real-time applications on edge computing platforms with remote attestation for security. *Proceedings of 2021 Ieee/acm Symposium on Edge Computing*, pages 403–408, 2021.

[RPC20] Niklas Reusch, Paul Pop, and Silviu S. Craciunas. Work-in-progress: Safe and secure configuration synthesis for tsn using constraint programming. *Real-time Systems Symposium (rtss), 2017 Ieee*, 2020-:387–390, 2020.

[RVBW06] Francesca Rossi, Peter Van Beek, and Toby Walsh. *Handbook of constraint programming*. Elsevier, 2006.

[RZCP20] N. Reusch, L. Zhao, S. S. Craciunas, and P. Pop. Window-based schedule synthesis for industrial IEEE 802.1Qbv TSN networks. In *Proc. WFCS*, 2020.

[RZH+19] Ju Ren, Deyu Zhang, Shiwen He, Yaoxue Zhang, and Tao Li. A survey on end-edge-cloud orchestrated network computing paradigms: Transparent computing, mobile edge computing, fog computing, and cloudlet. *ACM Comput. Surv.*, 52(6), oct 2019.

[SALC21] Ammad Ali Syed, Serkan Ayaz, Tim Leinmüller, and Madhu Chandra. Dynamic scheduling and routing for TSN based in-vehicle networks. In *Proc. ICC Workshops*, 2021.

[SAR+17] Anil Singh, Nitin Auluck, Omer Rana, Andrew Jones, and Surya Nepal. Rt-sane : Real time security aware scheduling on the network edge. *Ucc 2017 - Proceedings of The10th International Conference on Utility and Cloud Computing*, pages 131–140, 2017.

[SBCH13] Ahmad Sheikh, O. Brun, Maxime Chéramy, and Pierre-Emmanuel Hladik. Optimal design of virtual links in AFDX networks. *Real-Time Systems*, 49:308–336, 05 2013.

[SBHP11] W. Steiner, G. Bauer, B. Hall, and M. Paulitsch. TTEthernet: Time-Triggered Ethernet. In Roman Obermaisser, editor, *Time-Triggered Communication*. CRC Press, USA, Aug 2011.

[SDT+17] Eike Schweissguth, Peter Danielis, Dirk Timmermann, Helge Parzyjegla, and Gero Mühl. Ilp-based joint routing and scheduling for time-triggered networks. In *Proceedings of the 25th International Conference on Real-Time Networks and Systems (RTSN)*, page 8–17, New York, NY, USA, 2017. Association for Computing Machinery.

[SGAS20] Ali Shakarami, Mostafa Ghobaei-Arani, and Ali Shahidinejad. A survey on the computation offloading approaches in mobile edge computing: A machine learning-based perspective. *Computer Networks*, 182:107496, 2020.

[SGRT17] Fedor Smirnovt, Michael Gla, Felix Reimann, and Jürgen Teich. Optimizing message routing and scheduling in automotive mixed-criticality time-triggered networks. *Proceedings - Design Automation Conference*, 128280:48, 2017.

[SHHS03] J. Schmitt, P. Hurley, M. Hollick, and R. Steinmetz. Per-flow guarantees under class-based priority queueing. In *IEEE Global Telecommunications Conference*, pages 4169–4174, 2003.

[SHM+21] Youhwan Seol, Doyeon Hyeon, Junhong Min, Moonbeom Kim, and Jeongyeup Paek. Timely survey of time-sensitive networking: Past and future directions. *Ieee Access*, 9:142506–142527, 2021.

[Sho14] Adam Shostack. *Threat modeling: Designing for security*. John Wiley & Sons, 2014.

[Sin06] Oliver Sinnen. *Task Scheduling for Parallel Systems*. wiley, 2006.

[Sin07a] O. Sinnen. *Task scheduling for parallel systems*, volume 60. John Wiley & Sons, USA, 2007.

[Sin07b] Oliver Sinnen. *Fundamental Heuristics*, chapter 5, pages 108–144. John Wiley & Sons, Ltd, 2007.

[SKJ18] Sebastian Schriegel, Thomas Kobzan, and Jürgen Jasperneite. Investigation on a distributed sdn control plane architecture for heterogeneous time sensitive networks. In *Proc. WFCS*, 2018.

[SKR+18] Olena Skarlat, Vasileios Karagiannis, Thomas Rausch, Kevin Bachmann, and Stefan Schulte. A framework for optimization, service placement, and runtime operation in the fog. In *IEEE International Conference on Utility and Cloud Computing*, pages 164–173, 2018.

[SL16] Rodrigo Vieira Steiner and Emil Lupu. Attestation in wireless sensor networks: A survey. *ACM Comput. Surv.*, 49(3), September 2016.

[SM07] Jill Slay and Michael Miller. Lessons learned from the maroochy water breach. volume 253, pages 73–82, 03 2007.

[SNA+13] Ivan Studnia, Vincent Nicomette, Eric Alata, Yves Deswarte, Mohamed Kaaniche, and Youssef Laarouchi. Survey on security threats and protection mechanisms in embedded automotive networks. *Proc. DSN*, 2013.

[SNSH21] Khaled M. Shalghum, Nor Kamariah Noordin, Aduwati Sali, and Fazirulhisyam Hashim. Network calculus-based latency for time-triggered traffic under flexible window-overlapping scheduling (FWOS) in a time-sensitive network (TSN). *Applied Sciences*, 11(9), 2021.

[SOCS18] Ramon Serna Oliver, Silviu S. Craciunas, and Wilfried Steiner. IEEE 802.1Qbv gate control list synthesis using array theory encoding. In *Proc. RTAS*. IEEE, 2018.

[SOLM22] Thomas Stüber, Lukas Osswald, Steffen Lindner, and Michael Menth. A survey of scheduling in time-sensitive networking (tsn), 2022.

[SP18] Saad Zia Sheikh and Muhammad Adeel Pasha. Energy-efficient multicore scheduling for hard real-time systems: A survey. *ACM Trans. Embed. Comput. Syst.*, 17(6), dec 2018.

[SPX19] Weisong Shi, George Pallis, and Zhiwei Xu. Edge computing [scanning the issue]. *Proceedings of the IEEE*, 107(8):1474–1481, 2019.

[SS11] David J. Smith and Kenneth G.L. Simpson. *Safety Critical Systems Handbook*. Elsevier Ltd, 2011.

[Ste10] Wilfried Steiner. An evaluation of SMT-based schedule synthesis for time-triggered multi-hop networks. In *Proc. RTSS*, USA, 2010. IEEE.

[SWYS11] Jianhua Shi, Jiafu Wan, Hehua Yan, and Hui Suo. A survey of cyber-physical systems. In *2011 international conference on wireless communications and signal processing (WCSP)*, pages 1–6. IEEE, 2011.

[TDDFD20] Koen Tange, Michele De Donno, Xenofon Fafoutis, and Nicola Dragoni. A systematic survey of industrial internet of things security: Requirements and fog computing opportunities. *IEEE Communications Surveys & Tutorials*, 22(4):2489–2520, 2020.

[Tro22] Jakob Slyngborg Trolle. Leverandør lukkede it-system efter sikkerhedsbrist, og pludselig stod alle tog i danmark stille. *Danish Broadcasting Corporation*, 2022.

[TSP15] Domiţian Tămaş-Selicean and Paul Pop. Design optimization of mixed-criticality real-time embedded systems. *ACM Trans. Embed. Comput. Syst.*, 14(3), apr 2015.

[TSPS15] Domiţian Tămaş-Selicean, Paul Pop, and Wilfried Steiner. Design optimization of TTEthernet-based distributed real-time systems. *Real-Time Syst.*, 51(1):1–35, January 2015.

[vADF+20] Christian von Arnim, Mihai Drăgan, Florian Frick, Armin Lechler, Oliver Riedel, and Alexander Verl. Tsn-based converged industrial networks: Evolutionary steps and migration paths. In *Proc. ETFA*, volume 1, pages 294–301, 2020.

[VBHT22] Marek Vlk, Kateřina Brejchová, Zdeněk Hanzálek, and Siyu Tang. Large-scale periodic scheduling in time-sensitive networks. *Computers & Operations Research*, 137:105512, 2022.

[VHB+20] M. Vlk, Z. Hanzálek, K. Brejchová, S. Tang, S. Bhattacharjee, and S. Fu. Enhancing schedulability and throughput of time-triggered traffic in IEEE 802.1Qbv time-sensitive networks. *IEEE Transactions on Communications*, 68(11):7023–7038, 2020.

[VHT21] Marek Vlk, Zdeněk Hanzálek, and Siyu Tang. Constraint programming approaches to joint routing and scheduling in time-sensitive networks. *Computers & Industrial Engineering*, 157:107317, 2021.

[VW02] Stefan Voß and David L. Woodruff. *Optimization Software Class Libraries*. Springer US, Boston, MA, 2002.

[Wan06] E. Wandeler. *Modular performance analysis and interface-based design for embedded real-time systems*. Shaker, 2006.

[WH00] Bin Wang and J.C. Hou. Multicast routing and its qos extension: problems, algorithms, and protocols. *IEEE Network*, 14(1):22–36, 2000.

[Wol09] Wayne Wolf. Cyber-physical systems. *Computer*, 42(03):88–89, 2009.

[WT06a] E. Wandeler and L. Thiele. Optimal TDMA time slot and cycle length allocation for hard real-time systems. In *Proc. ASP-DAC*, 2006.

[WT06b] E. Wandeler and L. Thiele. Real-Time Calculus (RTC) Toolbox. `http://www.mpa.ethz.ch/Rtctoolbox`, 2006. Accessed: 31.12.2022.

[XJL+19] Yinhao Xiao, Yizhen Jia, Chunchi Liu, Xiuzhen Cheng, Jiguo Yu, and Weifeng Lv. Edge computing security: State of the art and challenges. *Proceedings of the IEEE*, 107(8):1608–1631, 2019.

[ZPC18] L. Zhao, P. Pop, and S. S. Craciunas. Worst-case latency analysis for IEEE 802.1Qbv time sensitive networks using network calculus. *IEEE Access*, 6:41803–41815, 2018.

[ZPGF32] L. Zhao, P. Pop, Z. J. Gong, and B. W. Fang. Improving latency analysis for flexible window-based GCL scheduling in TSN networks by integration of consecutive nodes offsets. *IEEE Internet of Things*, 2020, Early Access Article, https://doi.org/10.1109/JIOT.2020.3031932.

[ZPL+17] L. Zhao, P. Pop, Q. Li, J. Chen, and H. Xiong. Timing analysis of rate-constrained traffic in TTEthernet using network calculus. *Real-Time Systems*, 52(2):254–287, 2017.

[ZPS17] Luxi Zhao, Paul Pop, and Sebastian Steinhorst. Quantitative performance comparison of various traffic shapers in time-sensitive networking. *CoRR*, abs/2103.13424, 2017. `https://arxiv.org/abs/2103.13424`.

[ZQLY19] R. Zhao, G. Qin, Y. Lyu, and J. Yan. Security-aware scheduling for TTEthernet-based real-time automotive systems. *IEEE Access*, 7:85971–85984, 2019.

[ZSEP21a] Yuanbin Zhou, Soheil Samii, Petru Eles, and Zebo Peng. Asil-decomposition based routing and scheduling in safety-critical time-sensitive networking. In *Proc. RTAS*, 2021.

[ZSEP21b] Yuanbin Zhou, Soheil Samii, Petru Eles, and Zebo Peng. Reliability-aware scheduling and routing for messages in time-sensitive networking. *ACM Trans. Embed. Comput. Syst.*, 20(5), 2021.

[ZXZL14] L. X. Zhao, H. G. Xiong, Z. Zheng, and Q. Li. Improving worst-case latency analysis for rate-constrained traffic in the Time-Triggered Ethernet network. *IEEE Communications Letters*, 18(11):1927–1930, 2014.

[ZZJ+13] Xia Zhang, Jinyu Zhang, Wei Jiang, Yuexi Ma, and Ke Jiang. Design optimization of security-sensitive mixed-criticality real-time embedded systems. 2013.