FLEXIBILITY *in* REAL-TIME NETWORKS

Analytical Approaches for Adapting Time-Sensitive Networks to Dynamic Traffic Requirements

Vom Fachbereich Elektrotechnik und Informationstechnik
der Technischen Universität Darmstadt
zur Erlangung des akademischen Grades eines
Doktor-Ingenieurs (Dr.-Ing.)
genehmigte Dissertation

von

CHRISTOPH GÄRTNER, M.SC.

Vorsitz: Prof. Dr.-Ing. Jutta Hanson
Referent: Prof. Dr.-Ing. Dr. h.c. Ralf Steinmetz
Korreferent: Prof. Dr.-Ing. Amr Rizk

Tag der Einreichung: 7. Mai 2024
Tag der Disputation: 18. Juli 2024

Darmstadt 2024

Christoph Gärtner, M.Sc.: *Flexibility in Real-time Networks,*
Analytical Approaches for Adapting Time-Sensitive Networks to Dynamic Traffic Requirements

Darmstadt, Technische Universität Darmstadt

Jahr der Veröffentlichung der Dissertation auf TUprints: 2024
Tag der mündlichen Prüfung: 18. Juli 2024

Dieses Dokument wird bereitgestellt von / This document is provided by
tuprints, E-Publishing-Service der Technischen Universität Darmstadt.
http://tuprints.ulb.tu-darmstadt.de
tuprints@ulb.tu-darmstadt.de

Bitte zitieren Sie dieses Dokument als: / Please cite this document as:
URN: urn:nbn:de:tuda-tuprints-278806
URI: https://tuprints.ulb.tu-darmstadt.de/id/eprint/27880

Die Veröffentlichung steht unter folgender Creative Commons Lizenz:
Namensnennung 4.0 International (CC BY 4.0 International)
https://creativecommons.org/licenses/by/4.0/deed.de

This publication is licensed under the following Creative Commons License:
Attribution 4.0 International (CC BY 4.0 International)
https://creativecommons.org/licenses/by/4.0/deed.en

ABSTRACT

In the context of Industry 4.0, achieving flexible and deterministic data delivery is crucial, particularly within networks that leverage Time-Sensitive Networking (TSN) – an extension of traditional Ethernet – for real-time service guarantees. Current TSN configurations, however, are predominantly static and do not offer the adaptability required to cater to the continuously evolving demands of applications. Adaptations become increasingly complex when changes need to be incorporated during operation, and applications demand more stringent service guarantees. This complexity stems from elaborate TSN planning requirements, resulting in static switch configurations. Furthermore, these configurations lack indicators for the reconfiguration potential, thus making and predicting adaptations difficult. The static nature of TSN schedule configurations typically necessitates complete reconfigurations, resulting in service downtimes for applications in the network.

To address these challenges, this thesis introduces a novel flexibility metric called "flexcurve" designed to enhance the management of dynamic TSN networks. The introduction of the flexcurve metric aims to quantify the flexibility at network bottlenecks by considering frame size and other traffic requirements, thereby measuring possibilities for the integration of new traffic without extensive rescheduling. Consequently, it enables a central TSN controller to perform more informed traffic admissibility checks, path selection, and scheduling adjustments during the configuration planning phase. Utilizing a path-based approach, the flexcurve metric captures the essence of end-to-end scheduled traffic. Furthermore, the formulation allows for both disaggregation and aggregation, enhancing efficiency. Disaggregations enable traffic admissibility checks of different sources simultaneously, while aggregations facilitate the rapid construction of the metric from existing configurations.

The thesis builds on the flexcurve metric and introduces a search heuristic algorithm that employs the flexcurve for dynamic scheduling. This algorithm is designed to generate eligible candidates and incorporates a secondary heuristic for pruning to select the most promising candidates. Additionally, methods are included for choosing paths that optimize the value of the flexcurve.

As final contributions, the deployment capabilities of TSN to non-TSN hardware have been augmented by leveraging programmable Push-In-First-Out (PIFO) queues and by developing the Residence Delay Aggregation (RDA) method, which ensures per-flow delay guarantees on programmable switches. Additionally, the flexcurve has been further extended by integrating multi-mechanism support, significantly improving network flexibility and broadening TSN deployment opportunities.

KURZFASSUNG

Industrie 4.0 erfordert den Einsatz flexibler und deterministischer Datenübertragung, besonders in Netzen, die Time-Sensitive Networking (TSN) nutzen. TSN, eine Erweiterung von Ethernet, bietet Echtzeitgarantien für die Datenübertragung. Allerdings sind bestehende TSN-Konfigurationen überwiegend statisch und können sich nicht ausreichend an sich dynamisch ändernde Anforderungen anpassen. Ihre statische Natur, verbunden mit einem hohen Planungsaufwand und fehlenden Indikatoren für Rekonfigurationsmöglichkeiten, erschwert die Anpassung an veränderliche Bedingungen und führt bei Änderungen häufig zu Netzausfällen.

Um diese Herausforderungen anzugehen, wird in dieser Arbeit eine neuartige Flexibilitätsmetrik, die "Flexcurve" eingeführt, die es ermöglicht, das Management von dynamischen TSN-Netzwerken zu verbessern. Ziel der Flexcurve-Metrik ist es, die Flexibilität an Netzengpässen zu quantifizieren, indem sie die Datengrößen und andere Verkehrsanforderungen von Anwendungen berücksichtigt. Dadurch können ohne umfangreiche Neuplanung die Möglichkeiten für die Integration neuer Datenströme gemessen werden. Folglich ermöglicht sie es einem zentralen TSN-Controller, Prüfungen über die Zulässigkeit von Datenströmen, die Auswahl von geeigneten Pfaden und eine Anpassung der Zeitplanung durchzuführen. Durch einen pfadbasierten Ansatz erfasst die Flexcurve-Metrik den Ende-zu-Ende zeitgeplanten Netzverkehr. Darüber hinaus erlaubt die konkrete Formulierung der Flexibilitätsmetrik sowohl ein "Aufbrechen" als auch ein "Zusammensetzen", was die Effizienz steigert. Das Aufbrechen ermöglicht Prüfungen auf Zulässigkeit von verschiedenen Quellen gleichzeitig, während das Zusammensetzen den schnellen Aufbau der Metrik aus bestehenden Konfigurationen erleichtert.

Aufbauend auf dieser Metrik präsentiert diese Arbeit einen heuristischen Suchalgorithmus, der die Flexcurve für eine dynamische Zeitplanung verwendet. Dieser Algorithmus ist darauf ausgelegt, geeignete Zeitplan-Kandidaten zu erzeugen und beinhaltet eine sekundäre Heuristik zum Vorselektieren, um die vielversprechendsten Kandidaten auszuwählen. Zusätzlich sind Methoden zur Auswahl von Pfaden enthalten, die den Wert der Flexcurve optimieren.

Abschließend betrachtet diese Arbeit die Einsatzfähigkeit von TSN auf Nicht-TSN-Geräten durch die Nutzung programmierbarer Push-In-First-Out (PIFO)-Puffer sowie durch die Entwicklung der Residence Delay Aggregation (RDA)-Methode. RDA ermöglicht Latenzgarantien pro Datenstrom auf programmierbaren Netzwerkgeräten. Zudem wurde die Flexcurve durch Unterstützung für mehrere Mechanismen erweitert, was die Netzwerkflexibilität verbessert und die Einsatzmöglichkeiten von TSN erweitert.

PREVIOUSLY PUBLISHED MATERIAL AND USED TOOLS

This thesis includes material previously published in scientific journals and conferences. Table 1 summarizes all previously published material that was included as part of this thesis.

Scientific research is often the result of teamwork. This work reflects the contributions of computer scientists and researchers from other fields. I will clearly list the contributions and affiliations of all co-authors and contributors. If an affiliation isn't listed at the first mention, it means that the person is, or was, a colleague at the Technical University of Darmstadt (TU Darmstadt), working in the Multimedia Communications Lab (KOM).

Several figures and tables that illustrate fundamental concepts and outcomes have been reproduced in this thesis. Additionally, certain sections directly incorporate or closely paraphrase content from prior publications. Following, I detail the chapters that contain either direct quotations or adapted segments from previous published works or other collaborations. A comprehensive list of my publications, including those not directly related to this thesis, is given in Appendix C. Throughout this document, the pronoun *we* is used to highlight the collective nature of the research. In general, in terms of previous contributions, Prof. Dr.-Ing. Amr Rizk (University of Duisburg-Essen), Prof. Dr. Boris Koldehofe (Technical University of Ilmenau), Dr.-Ing. Ralf Kundel, and Prof. Dr.-Ing. Ralf Steinmetz have played vital roles in supervising the entirety of the research process, offering critical insights into the methodology, approach, and overall development. Specific acknowledgments are made to each person as necessary, should the contributions vary for a particular publication.

Beginning with **Sections 3.1 and 3.2**, I depict the thesis context and controller design. This essence is derived from the results of three prior publications: [30], [26], and [28]. Additionally, I incorporate the controller design from [27]. All four papers were created as part of project T3 within the German Research Foundation (DFG)-funded Collaborative Research Center (CRC) 1053 – MAKI.

In [30], we first proposed the flexcurve concept, which is an integral part of this thesis. Prof. Dr.-Ing. Amr Rizk and Prof. Dr. Boris Koldehofe supported me in developing the underlying concept of the paper. Dr. René Guillaume (Robert Bosch GmbH) provided feedback on the underlying concept and offered a practical context from an industrial perspective. Dr.-Ing. Rhaban Hark (ABB AG) supervised the entire process and provided valuable feedback regarding methodology and approach. We extended the concept in the subsequent paper [26] to include deadline-awareness. Additionally, the paper presents a search algorithm for finding flow embeddings that can also be utilized to construct a deadline-aware flexcurve. Amr Rizk, Boris Koldehofe, and Ralf Kundel assisted in extending the concept and participated in the writing process. René Guillaume provided valuable feedback on the overall conceptualization and contributed further knowledge on the industrial context. We revised this paper with an

Table 1: Previously published material.

Section	[30]	[26]	[28]	[29]	[100]	[27]
Chapter 3: FLEXIBILITY-BASED TSN MANAGEMENT						
3.1: Managing Scenario	✓	✓				✓
3.2: Controller	✓	✓	✓			✓
Chapter 4: FLEXIBILITY NOTION						
4.1: Flexcurve: A Notion of Flexibility for TSN	✓					
4.1.1: Disaggregations for Admissibility Decisions	✓					
4.1.2: Aggregations for Quick Construction and Incremental Updates		✓	✓			
4.1.3: Deadline-awarenes		✓	✓			
4.2: Evaluation	✓	✓	✓			
Chapter 5: OPTIMIZATION						
5.1.1: Eligibility Candidates for Deadline-aware Flexcurves		✓	✓			
5.1.2: Eligibility Candidate Selection		✓	✓			
5.3: Queue Assignments			✓			
5.4: Evaluation		✓	✓			✓
Chapter 6: MULTI-MECHANISMS						
6.1: PIFO Structures in TSN				✓		
6.2: RDA: Residence Delay Aggregation					✓	
Appendix A: APPENDIX						
A.1: Flexibility-aware Controller Implementation						✓
A.2: Lemmas and Proofs			(✓)			

extended version in [28], where we include a fast candidate selection of results from the previous search algorithm and queue assignments to enable deployment on actual TSN hardware switches. The share of contributions of this paper are distributed as in [26]. For the demonstration in [27], all co-authors provided feedback on the paper writing process.

Continuing with **Chapter 4**, I included concepts and results from [30], [26], and [28] once again. In particular, Section 4.1 presents the flexcurve concept, first introduced in [30], along with a similar example. In Section 4.1.1, the concept of flexcurve disaggregations is drawn from [30]. Furthermore, in Section 4.1.2, the concept of flexcurve aggregations is derived from [26] and [28]. Moreover, Section 4.1.3 incorporates the concepts of flow delays and deadline-aware flexcurves, as given in [26] and [28]. Lastly, in the chapter's evaluation, cf. Section 4.2, I depicted results from the flexcurve scheduler comparison, as detailed in [26] and [28].

Continuing with **Chapter 5**, in Section 5.1.1, I included the approach for finding eligibility candidates for deadline-aware flexcurves from [26, 28]. In Section 5.1.2, I incorporated Section 4.3.1 from [28], detailing the approach for selecting eligibility candidates verbatim. I adjusted this content to align with the notation used throughout this thesis. In Section 5.3, I included Section 4.2 from [28], which outlines the method for finding queue assignments, verbatim. This content has been adjusted to align with the notation used in the rest of this thesis. For the chapter's evaluation, cf. Section 5.4, I included results and assumptions from [26–28].

Continuing with **Chapter 6**, in Section 6.1, I included the paper [29], regarding PIFO usage for TSN, verbatim. I adjusted this content to align with the notation used throughout this thesis. Furthermore, in Section 6.2, I included the paper [100], regarding the proposed mechanism RDA, verbatim. I adjusted this content to align with the notation used throughout this thesis. In [29], we introduced the idea of using PIFO to deploy the Time Aware Shaper TSN mechanism. Amr Rizk assisted me in the conceptualization for this paper. Rhaban Hark supervised the entire process. René Guillaume provided help with the writing process. In [100], Chengbo Zhou and I are equal first authors. We developed the idea together based on similar original concepts, with me focusing on the flexibility aspects, and Chengbo Zhou focusing on the conceptualization for programmable hardware. Chengbo Zhou and I had equal share in the paper writing process. Ralf Kundel further assisted in the writing process. Amr Rizk provided further help for the overall conceptualization. Boris Koldehofe and Prof. Dr. Björn Scheuermann supervised the process.

Lastly for **Appendix A**, Appendix A.1 includes the controller design from [27]. Appendix A.2 includes the proof for Lemma A.3 given in [28] verbatim. I adjusted this content to align with the notation used throughout this thesis.

Tools used for creation of the thesis

This thesis is the result of my independent work, it has been in parts linguistically revised with the help of the tools Grammarly and ChatGPT. These tools were employed solely for the purpose of revising grammatical structure and ensuring clarity of expression.

CONTENTS

1 INTRODUCTION 1
- 1.1 Motivation for Flexibility in Time-Sensitive Networking . . . 2
- 1.2 Research Challenges . . . 3
- 1.3 Research Goals and Contributions . . . 4
- 1.4 Structure of the Thesis . . . 6

2 BACKGROUND & RELATED WORK 7
- 2.1 Notational Reference . . . 7
- 2.2 Time-Sensitive Networking . . . 10
- 2.3 Scheduling . . . 15
- 2.4 Software-defined Networking . . . 19
- 2.5 Flexibility . . . 20
- 2.6 Transitions in Time-Sensitive Networking . . . 23

3 FLEXIBILITY-BASED TSN MANAGEMENT 25
- 3.1 Managing Scenario . . . 25
  - 3.1.1 Network Model . . . 28
- 3.2 Controller . . . 29

4 FLEXIBILITY NOTION 33
- 4.1 Flexcurve: A Notion of Flexibility for TSN . . . 35
  - 4.1.1 Disaggregations for Admissibility Decisions . . . 40
  - 4.1.2 Aggregations for Quick Construction and Incremental Updates . 44
  - 4.1.3 Deadline-awarenes . . . 47
  - 4.1.4 Holistic Flexibility View . . . 50
- 4.2 Evaluation . . . 52

5 OPTIMIZATION 57
- 5.1 Flexcurve-based Scheduling . . . 57
  - 5.1.1 Eligibility Candidates for Deadline-aware Flexcurves . . . 57
  - 5.1.2 Eligibility Candidate Selection . . . 62
  - 5.1.3 In-place Scoring . . . 64
- 5.2 Path Selection . . . 65
- 5.3 Queue Assignments . . . 66
- 5.4 Evaluation . . . 68

6 MULTI-MECHANISMS 77
- 6.1 PIFO Structures in TSN . . . 77
- 6.2 RDA: Residence Delay Aggregation . . . 81
- 6.3 Flexibility of Simultaneous usage of TSN mechanisms . . . 89

7 SUMMARY, CONCLUSIONS, AND OUTLOOK 95
7.1 Summary of the Thesis 95
7.1.1 Contributions 95
7.1.2 Conclusions 97
7.2 Outlook 98

BIBLIOGRAPHY 101

A APPENDIX 111
A.1 Flexibility-aware Controller Implementation 111
A.1.1 SMT Scheduler 112
A.2 Lemmas and Proofs 115
A.3 Additional Figures 118
A.4 In-place Scoring Scenarios 121
A.5 List of Acronyms 123

B SUPERVISED STUDENT THESES 125

C AUTHOR'S PUBLICATIONS 127

D ERKLÄRUNGEN LAUT PROMOTIONSORDNUNG 129

1 INTRODUCTION

Packet-switched computer networks are the foundation of modern digital communication, enabling data exchanges worldwide. They are indispensable in supporting data communication across multiple platforms, from the Internet to data centers, even to smartwatches. The evolution of network technologies, from early analog systems to modern digital networks, has been a journey of increasing complexity and capability. In packet-based communication, as in today's Ethernet-based computer networks, data packets are forwarded along their paths by network devices such as switches and routers. *Switched Ethernet* (IEEE Std 802.1Q [42]) is nowadays the fundamental technology for local networks, facilitating packet transmission. Ethernet devices are cheap and offer high performance.

Traditional switched Ethernet does not support service guarantees in data transmission. Packets adhere to a best-effort (BE) delivery, where each packet is forwarded as efficiently as possible, given the current condition and hardware capabilities. However, best-effort delivery is insufficient in many environments. In particular, industrial or in-vehicle communication requires real-time service guarantees for their applications to function [10, 11, 65, 96]. Here, additional measures are necessary to guarantee the traffic requirements of critical applications.

Historically, computer networks supporting real-time communication are built on proprietary systems. They are specifically designed for applications where reliable and timely delivery of data is critical [96]. A class of those systems are proprietary Ethernet-based solutions such as TTEthernet [93] and Profinet [76]. They emerged to meet the growing requirements of industrial communications [16, 21, 78, 96], thus showing the need for specialized solutions in environments where traditional Ethernet's best-effort service model falls short. Ethernet's ubiquity, interoperability, and scalability are desired.

This has led to the development and extension of IEEE Std 802.1Q switched Ethernet called Time-Sensitive Networking (TSN). The extensions provide mechanisms within standard Ethernet to handle time-sensitive data with the required service guarantees. This standard inclusion enables a diverse vendor ecosystem wherein devices are more likely to be interoperable. It also creates more accessible opportunities for research.

As larger-scale, e. g., factory-wide, networks become possible through the adoption of Ethernet, varying application requirements and evolving topologies pose new challenges. Although TSN is highly effective for its intended applications, the usage of TSN reduces the inherent flexibility of traditional Ethernet. Without TSN, Ethernet is quickly able to adapt to changes in topology and traffic, as there is no need to consider given service guarantees. This newfound rigidity in Ethernet systems can limit the network's scalability and adaptability in dynamic environments where network demands are continuously changing.

1.1 MOTIVATION FOR FLEXIBILITY IN TIME-SENSITIVE NETWORKING

As industries and processes evolve, the need for a network infrastructure that handles real-time requirements and adapts to changing conditions becomes paramount. For instance, in a factory environment, it is crucial to adjust network paths, accommodate new applications and their devices, or respond to changing production demands without impacting real-time communication. Through TSN and Ethernet, deploying factory-wide real-time traffic mixed with non-real-time traffic becomes possible. However, any change to the configuration is potentially affecting all currently deployed devices, highlighting the key limitation for flexible TSN-based networks. In the worst case, breaking the given guarantees leads to catastrophic failures. A TSN configuration is deployed on network devices, affecting the network's behavior to achieve the required communication requirements. The parameters for the configuration differ depending on which TSN mechanisms are available and needed.

Current configurations in TSN are typically static and designed for the existing network applications. A configurator collects application requirements beforehand and sets the network configuration accordingly. Creating a working configuration requires a thorough analysis of the requested requirements and the current configuration state. State-of-the-art approaches usually use constraint-based solvers like satisfiability modulo theories (SMT) or integer linear programming (ILP) for strict traffic requirements that are supported by scheduled traffic (ST) [13, 18]. However, these approaches are very time-intensive, even for light network utilizations. The resulting configuration also usually disregards potential future changes. This time-consuming computation and the resulting static configuration dramatically limit application adaptation possibilities once a configuration is deployed.

While less strict real-time requirements may offer more flexibility when more inherently flexible mechanisms can be used, this diminishes with stricter demands and increasing traffic volume.

> **Definition —** *Scheduled Traffic* (ST), also called time-triggered traffic, is a traffic class where network traffic is forwarded according to a precomputed timetable at each hop. This timetable defines the achieved bandwidth, delay, jitter, and sending period.

This configuration approach, while effective in the short term, poses significant limitations:

1. The static nature of these configurations limits online reconfiguration capabilities. Networks may require complete reconfiguration to implement changes, leading to potential operational disruptions or costly shutdowns when networks are reconfigured.
2. Configurations, where application requirements can be incorporated on the fly, are prohibited. Flexible configurations would allow for guaranteed adjustments

to accommodate unexpected or planned changes in device usage while not affecting previously deployed applications.

3. The inflexibility of current TSN configurations poses challenges in aligning with Industrial Internet of Things (IIoT) and Industry 4.0 goals for data exchange, requiring flexibility and determinism in communication [66].

The need to balance the deterministic nature of TSN with the desired inherent flexibility of traditional Ethernet is becoming increasingly critical as we move towards more interconnected and versatile network infrastructures. Approaching the versatility of traditional Ethernet with deployed TSN is a central goal of this thesis.

1.2 RESEARCH CHALLENGES

To clearly specify the meaning of flexibility in the context of TSN, we provide the following definition:

> **Definition** — *Flexibility* is the ability to accommodate future changes of configurations at runtime.

Building on the previously mentioned motivation to gain flexible TSN configurations. We identify three major challenges, that hinder an inherent flexbility in TSN:

Challenge: *Deployed TSN configurations can be complex and provide no inherent intuition for reconfiguration possibilities.*

Existing TSN configurations and their static nature often overlook future traffic changes, as the configuration is primarily created to satisfy the given application requirements. Consequently, determining feasible changes and their impact on active applications is non-trivial. For instance, in scheduled traffic environments, without rescheduling, it is unknown if new traffic can be included. Capturing the level of flexibility for the current configuration state can help determine whether traffic can be admitted. Further, this can also facilitate a more flexible TSN usage by allowing partial reconfigurations and direct adaptation to new applications. However, to be able to reason about flexibility, a flexibility metric is required that can assess the level of flexibility for TSN scheduled traffic.

Challenge: *Elaborate configuration planning requirements with strict real-time traffic.*

While the planning process of TSN configurations varies between mechanisms, traffic with strict requirements supported by scheduled traffic, require an elaborate scheduling phase. With scheduled traffic, packets are scheduled in advance at each network hop, considering interfering traffic, forwarding delays, and application requirements within the scheduler. The planning of such schedules often uses constraint satisfaction tools which impose high computational overhead. Heuristic methods can improve the scheduling speed significantly but result in less capable schedules. However, both

planning methods usually disregard flexibility aspects to support future changes. Minimizing the need for complete reconfigurations and improving the quality of TSN configurations requires an understanding of how knowledge of flexibility can support planning reconfigurations.

Challenge: *Mechanism selection and abstraction disparities.*

Deploying a configuration for real-time network applications necessitates a two-step process: 1. choosing an available and suitable mechanism that can support all service requirements, and 2. setting the mechanism's parameters to ensure all requirements are met. Often, there is a disparity between an application's requirements and the parameters of the chosen mechanism. Both tasks are non-trivial, as the selection of mechanisms and parameters significantly impacts the flexibility that can be achieved. It is beneficial for configuration flexibility to extend the variety of mechanisms and allow for their combined utilization. When multiple mechanisms are used, mechanisms can directly cater to their application requirements without the need to over-provision resources for service guarantees that are not needed. However, this requires the appropriate coordination in setting up the TSN configurations for multiple mechanisms. Knowledge of other mechanisms' attributes and their proper integration is needed within the configuration planning process to consider flexibility impacts properly.

1.3 RESEARCH GOALS AND CONTRIBUTIONS

The main goal of this work is to increase the flexibility of deployed TSN configurations for runtime adaptions. This is divided into the following primary research goals.

Research Goal 1: *Provide a mechanism for analyzing flexibility of scheduled traffic configurations.*

The flexibility level of the current state of configuration needs to be assessed to achieve increases in flexibility. In research goal 1, we focus this notion on scheduled traffic scenarios, which enable applications with strict real-time traffic requirements. These scenarios are highly relevant to industrial manufacturing applications and provide the least inherent flexibility due to the rigid precomputed schedule. To address this goal, we provide a flexibility notion for TSN schedules [26, 28, 30] that captures possibilities for future traffic admissions without interference to previously admitted scheduled traffic.

Research Goal 2: *Methods for optimizing the flexibility of TSN schedules.*

Building on the flexibility notion for TSN schedules, we aim as part of the 2nd research goal at improving scheduled traffic flexibility under constant changes. Reconfigurations are triggered by changing applications or changing application requirements participating in the network. Without respecting potential future changes, created configurations may become inadequate or require a complete re-scheduling, resulting in undesirable downtimes. Respecting potential future changes helps to reduce

this. The flexibility notion of Research Goal 1 lends itself to be combined with schedulers in the planning process to influence TSN configurations for increased flexibility. We incorporate this flexibility notion within flow scheduling procedures in [26–28]. Increasing schedule flexibility helps to reduce the need for an elaborate configuration re-scheduling.

Research Goal 3: *Enabling a flexible deployment of TSN mechanisms to achieve application requirements.*

Extending the variety of mechanisms, e. g., due to different hardware capabilities, creates more deployment opportunities, enabling more flexible device usage. We propose alternative mechanisms in [29] and [100]. Furthermore, concurrently using multiple mechanisms also leads to more flexible deployments. Selecting the appropriate mechanism for each application to prevent over-provisioning network resources is a straightforward way to maintain flexibility. To allow for multiple mechanisms while preserving flexibility, we have incorporated multi-mechanism support in the configuration planning process, extending our TSN flexibility notion, as detailed in Section 6.3.

Scope of the Thesis

In the following, we briefly discuss topics that fall outside the scope of this thesis. This thesis adopts the centralized Software-defined Networking (SDN) paradigm. A centralized global view is possible and common in local networks where TSN is commonly deployed. This means distributed reservation protocols, which are also part of TSN standardization, are not considered. Narrowing this focus also excludes layer three routed networks, as advanced by the IETF working group DetNet[1]. Resilience aspects of improving flexibility, either in a cyber security or redundancy sense, are also not considered. Except for Chapter 6, which considers multiple mechanisms, this thesis primarily examines approaches that require scheduled traffic. Scheduled traffic is a major focus for this thesis because of its inherent lack of flexibility due to enabling strict real-time guarantees and the necessity to precompute timetables. This choice is based on the expectation that enhancing the flexibility of scheduled traffic in TSN may yield significant benefits for its users and may also translate to approaches with less strict needs.

1 Information about the DetNet working group can be found here: https://datatracker.ietf.org/wg/detnet/about/ (Last accessed on 19th January 2024)

1.4 STRUCTURE OF THE THESIS

The structure of this thesis is outlined as follows: Chapter 2 provides background information and a review of related work relevant to the context of the thesis. Chapter 3 discusses our assumptions for a flexibility-aware TSN management. The first, second, and third research goals are addressed in Chapter 4, Chapter 5, and Chapter 6, respectively. Each of these chapters includes specific evaluations relevant to its focus. The thesis concludes with Chapter 7, which summarizes the contributions and provides an outlook on potential future work.

2 BACKGROUND & RELATED WORK

In this chapter, we provide relevant background knowledge and information on related work of this thesis. The subsequent section also provides with Table 2.1 descriptions for important notational symbols used throughout the remainder of this thesis.

2.1 NOTATIONAL REFERENCE

Table 2.1: Notational Reference

Symbol		Description
i, j	:	Natural number iterators within a local scope
$[i]$	:	Subset of natural numbers $\{1, \ldots, i\}$
$\lvert\cdot\rvert$	:	Cardinality of a set
G, V, E	:	Graph G with the set of nodes V and directed links E
v_i	:	The i-th node in V
p	:	A directed link or source port (v_i, v_j), the source port is associated to the node v_i
p_ω	:	The ω-th port in the path
P	:	A path consisting of a sequence source ports $(p_1, \ldots, p_m)$
m	:	Number of ports in the path
$F, \mathbf{F}$	:	Set of all flows F, Set of requested flows $\mathbf{F} = \{\mathbf{f}_1, \ldots, \mathbf{f}_u\} \subseteq F$
f_δ	:	The δ-th flow in F
u	:	Number of requested flows
α	:	Iterator of requested flows $\alpha \in \{1, \ldots, u\}$
$\mathbf{P}, \mathbf{P}_\alpha$	:	The requested path when $u = 1$ and α-th requested path
$\mathcal{P}$	:	Intersection of all requested source/destination port pairs: $\mathcal{P} = \bigcap \mathbf{P}_\alpha$
$\mathbf{f}, \mathbf{f}_\alpha$	:	The requested flow when $u = 1$ and α-th requested flow
$\mathfrak{P}_\delta$	:	Set of eligible paths for flow f_δ
h	:	Hyperperiod also known as network cycle period
$c, \mathbf{c}, \mathbf{c}_\alpha$	:	Frame size, requested frame size and α-th requested frame size

Continued on next page

Table 2.1 – continued from previous page

Symbol		Description
s_p	:	Port schedule for port p
φ_p	:	Duration of the last gap within port schedule s_p
$\vec{s}_p$	:	Port schedule s_p with end-shifted departure times
χ_p	:	Number of distinct frame departure times in s_p
$\mathcal{T}_{p,\beta}$	:	Time point of the β-th free slot at port schedule $\vec{s}_p$
n, κ	:	Natural numbers used as slot index iterators
d, $\mathbf{d}$	:	Frame deadline, requested frame deadline
λ	:	The λ-th flexcurve disaggregation step
$c_{max,P}$	:	Maximum frame size along path P
$\mathbf{c}_{max,\lambda,\mathcal{P}}$	:	Maximum frame size for the λ-th disaggregated flexcurve with $\mathbf{c}_{max,0,\mathcal{P}} = c_{max,\mathcal{P}}$
$\mathcal{C}_\lambda$	:	Set of requested indices $\mathcal{C}_\lambda \subseteq \{1, \dots, u\}$ that are admissible for the λ-th disaggregation
Ψ_p	:	Number of gaps within the port schedule s_p
$\vec{\Psi}_p$	:	Number of gaps within the end-shifted port schedule $\vec{s}_p$
$\vec{\Psi}^P$	:	Total number of end-shifted gaps along path P: $\vec{\Psi}^P = \sum_{p \in P} \vec{\Psi}_p$
g_p	:	Sequence of gap starting times $g_p = (g_p^1, g_p^2, \dots, g_p^{\Psi_p})$ for the port schedule s_p ordered by time relative to beginning of the schedule
$\vec{g}_p$	:	Sequence of gap starting times for the end-shifted schedule $\vec{s}_p$
Δ_p	:	Sequence of gap durations $\Delta_p = (\Delta_p^1, \Delta_p^2, \dots, \Delta_p^{\Psi_p})$ corresponding to g_p
$\vec{\Delta}_p$	:	Sequence of gap durations for the end-shifted schedule $\vec{s}_p$
a_ω	:	Assignment for a specific time-point within the port-schedule s_{p_ω}. E. g., a port transmission time for flow candidates
$\text{Gap}(a_\omega)$	:	The mapping from a port assignment a_ω to a specific gap is given by the function $\text{Gap}(a_\omega) = i, \quad$ where i is the largest index such that: $g_{p_\omega}^i \leqslant a_\omega$
q_ω	:	A specific candidate assignment a_ω
A	:	A sequence of assignments $A = (a_1, \dots, a_m)$
$\overline{\mathcal{A}}$	:	Set of all eligible assignments A for a flow with frame size c and deadline d

Continued on next page

Table 2.1 – continued from previous page

Symbol		Description
$\overline{\mathcal{A}}^{p}$	:	Reduces $\overline{\mathcal{A}}$ to a single port p_ω such that $a_\omega \in \overline{\mathcal{A}}^{\omega}$
$\overline{\overline{\mathcal{A}}}$	:	Set of eligible assignments A derived from Algorithms 2 and 3 for deadline and frame size constraints
$\overline{\overline{\mathcal{A}}}^{p}$	:	Reduces $\overline{\overline{\mathcal{A}}}$ to a single port p_ω such that $a_\omega \in \overline{\mathcal{A}}^{\omega}$
$\overline{\overline{\mathcal{A}}}_A$	:	The set $\overline{\overline{\mathcal{A}}}$ pruned with all overlaps of A
$t(n, \kappa)$	:	The delay between two time slots n, κ for any two consecutive ports $(p_\omega, p_{\omega+1})$ for a given frame size of c and hyperperiod h
t_{max}	:	The maximum delay between two consecutive ports is given as $h + c - 1$ for a given frame size of c and hyperperiod h
$T(A)$	:	The end-to-end delay for a given assignment A
$\max(T)$	:	The maximum end-to-end delay is given as $\max(T) = (m - 1)t_{max} + c$ for a given path length m, frame size c and hyperperiod h
$\mathcal{Q}$	:	Set of shared queue identifiers available for scheduled traffic in the network
$\hat{\mathcal{Q}}_\omega$	:	The set of occupied queue identifiers at port p_ω at time point $a_{\omega-1}$ for the duration of the frame c
$\mathcal{Q}'_\omega$	:	The set of available queues at port p_ω is given by $\mathcal{Q}'_\omega = \mathcal{Q} \cap \hat{\mathcal{Q}}_\omega$
$C_p(n)$	:	Cumulative capacity up to slot index n at port p
$b_P(c)$	:	Basic flexcurve value for frame size c along path P
$b_P^d(c)$	:	Deadline-aware flexcurve value for frame size c, deadline d along path P
$\tilde{b}_\Delta(c)$	:	Canonical flexcurve value for frame size c and period Δ
w	:	A Credit-based Shaper (CBS) class $w \in \{1, \dots, 8\}$
cd_p^w	:	The CBS deadline for port p and class w
ed_p^w	:	The CBS eligibility duration for port p and class w

2.2 TIME-SENSITIVE NETWORKING

Time-Sensitive Networking (TSN) is the Ethernet standard extension designed to integrate real-time communication into switched Ethernet. Available migration strategies, such as those outlined in [85] for the proprietary Ethernet system ProfiNet, simplify the transition to this standard protocol. The IEEE Standard 802.1Q [42] has incorporated several extensions over the years (e. g., IEEE Std 802.1Qbv [40], 802.1Qbu [41], 802.1Qci [45], 802.1Qcr [43]) to support real-time data communication.

We will next define the term *real-time* as used in this thesis.

> **Definition** — *Real-time* in the context of data transmission can give service guarantees for data delivery in the network.

TSN consists of several sub-standards, including standards,

- **IEEE Std 802.1Q** [42]: *Bridges and Bridged Networks*
- **IEEE Std 802.1AS** [47]: *Timing and Synchronization for Time-Sensitive Applications*
- **IEEE Std 802.1CB** [46]: *Frame Replication and Elimination for Reliability*

that constitute the core functionality. We define *Flows* in the context of this thesis as:

> **Definition** — A *flow* is a sequence of Ethernet frames that adhere to application specific requirements.

Flows are the main subject of TSN, as additional real-time mechanisms are deployed to ensure the compliance of application requirements for application flows. Flows are identifiable, e. g., by utilizing the unique parameters such as source and destination, or using a unique identifier field. The TSN standard also includes a configuration model. We describe in detail in Chapter 3, how we designed and apply a flexibility-based configuration model for TSN management.

Delays in Switched Forwarding

The transmission of flows in a switched communication networks is not instantaneous. Different delays are introduced as the flow's frames are forwarded. The end-to-end delay of frames is comprised of *transmission*, switch *processing*, port *queueing*, and link *propagation* delays. The transmission delay is the time required for a frame to be transmitted. This time is influenced by the line rate at which the frame is transmitted. The processing delay is the time required for the reception of the frame at the forwarding element until it is available in the egress queue. The queueing delay is the time a frame resides in the egress queue until it is eligible for forwarding. The propagation delay is the delay introduced by the physical layer until the first bit arrives at the ingress port. It depends on the forwarding hardware, length and the type of the medium.

Time Synchronization

Some TSN mechanisms require a globally synchronized time to function. To achieve synchronization between time-aware end-devices and switches (Ethernet bridge), the IEEE Std 802.1AS [47] can be used, and is recommended in IEEE Std 802.1Q. It is based on the Precision Time Protocol (IEEE Std 1588 [39]), and provides configurations and procedures to ensure synchronization for time-sensitive applications as supported by IEEE Std 802.1Q [42] can be met. In this thesis, we assume that the time synchronization is given, and we will not consider the impacts of failures, interferences, or inaccuracies that may arise from the used time synchronization method. Thus, we will not present further details regarding the time synchronization methods.

TSN Mechanisms

The TSN standard collection encompasses multiple mechanisms. Depending on the flow requirements, the appropriate mechanisms need to be chosen. In the following Table 2.2, we first provide a quick overview of the main traffic shaping mechanisms specified currently in IEEE Std 802.1Q, followed by a more detailed description of the relevant mechanisms for this thesis. The possible guarantees any mechanism can give, also depend on the deployed configuration, e. g., bandwidth parameters.

Synchronous: Scheduled Traffic

IEEE Std 802.1Q [42, p. 222] specifies enhancements for scheduled traffic, also known as the Time Aware Shaper (TAS). This extension enables switch egress queues to be scheduled based on time with cyclic gate control lists (GCLs). The egress queues are assumed to adhere to a first-in-first-out (FIFO) principle. The standard allows for a total of eight egress queues per port. The Priority Code Point (PCP) header field (3-bit field) of the VLAN tag is used to distinguish between frame priority values, to enqueue the frame in the corresponding queue. Consequently, in order to be effective, global time synchronization is required, which allows for synchronized execution of the network's GCLs (cf. Figure 2.1). A single GCL is utilized per egress port, controlling the eligibility of a port's queues for frame transmission selection.

Queued frames from each port's queues are selected by a transmission selection algorithm, if the corresponding queue's gate is open. This algorithm defaults to strict priority, meaning the queue with the highest priority is selected. However, alternative selection algorithms exist: credit-based shaping, asynchronous traffic shaping, or enhanced transmission selection. With the freely configurable GCLs and several transmission selection algorithms, the TSN enhancements for scheduled traffic can be applied in a variety of ways.

Each entry of the GCL, controlling the queue's eligibility states, is executed at its specified time. The GCL execution is repeated after a configurable cycle time. Additionally, the base time from which future cycles are computed is also configurable. While port cycles can be configured independently, for ease of configuration, it is com-

Table 2.2: TSN Traffic Shaping Mechanism Overview [42]

	Time Sync.	Complexity
Strict Priority Prioritizes network packets based on VLAN header priority.	✗	Low
Enhanced Transmission Selection Allocates the available bandwidth among different traffic classes, e.g., using variants of round-robin scheduling.	✗	Low
Credit-based Shaper Manages the bandwidth allocation of affected traffic classes through a credit system to reduce congestion and ensure fairness.	✗	Medium
Asynchronous Traffic Shaper Similar to the credit-based shaper, allows per flow delay modeling.	✗	Medium
Scheduled Traffic Can ensure precisely timed packet delivery via time-synchronized queue scheduling.	✓	High
Cyclic Queueing and Forwarding The affected traffic is transmitted and queued in a cyclic fashion. Allows for simple flow latency calculations.	✓	Low

mon to apply a network cycle time or hyperperiod, h. This hyperperiod can be selected by computing the least common multiple of the periods of all given flows.

Scheduled traffic is the main mechanism that is used as underlying TSN mechanism in thesis. Unless otherwise specified, we assume the default strict priority transmission selection when scheduled traffic is employed. The *traffic class* scheduled traffic, as introduced, is defined as follows:

> **Definition** — *Scheduled Traffic* (ST), also called time-triggered traffic, is a traffic class where network traffic is forwarded according to a precomputed timetable at each hop. This timetable defines the achieved bandwidth, delay, jitter, and sending period.

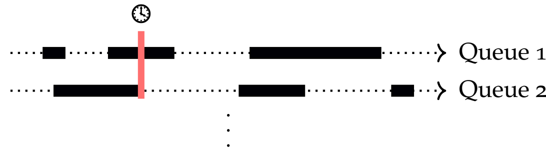


Figure 2.1: Execution of gate opening events is time synchronized across the network. Figure derived from [28].

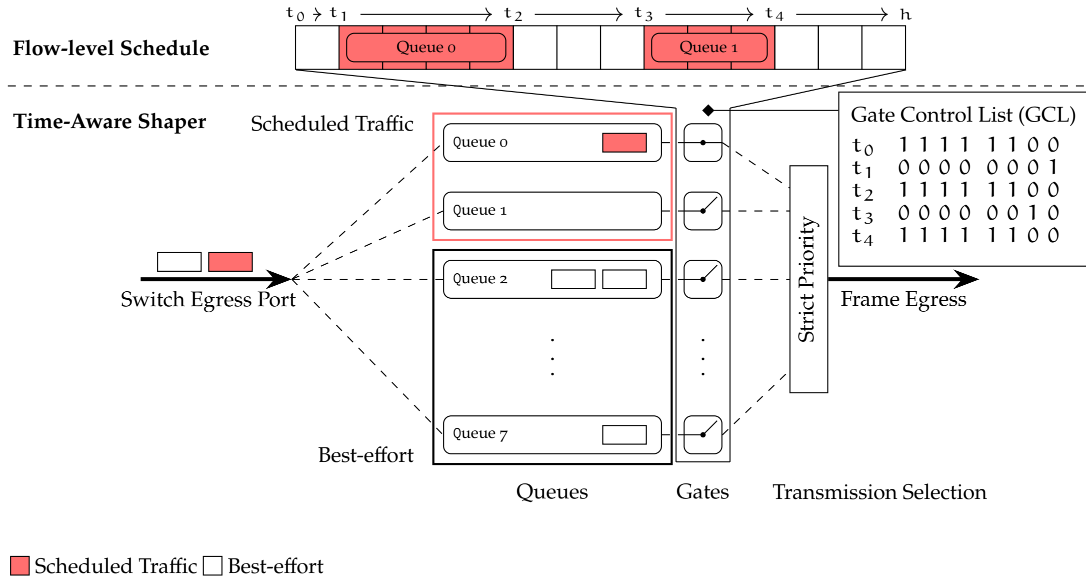


Figure 2.2: Example mapping from flow- to queue-level schedule for a single egress port. Events within the flow-level schedule can be mapped to gate events in the GCL. Best-effort and scheduled traffic frames are queued in their corresponding queues. The gates are depicted with state t_1. Figure derived from [28].

With its flexible configuration, scheduled traffic configuration can be complex and focus on different goals, with different approaches to create such a configuration. We summarize some state-of-the-art approaches in Section 2.3.

While TSN supports queue-level schedules via GCLs directly, scheduled traffic can also apply at the flow-level. That is, each flow's frame is forwarded according to a precomputed timetable. We can map flow-level schedules to queue-level gate opening and closing events, provided that the availability of queues permits this mapping. Time is assumed to be discrete, as the time granularity within hardware is limited. For instance, a time granularity of 1ns results in discrete time slots, each of 1ns duration. The number of slots that are required depend on the port's line rate and overall flow reservation requirements. We depict an example mapping from flow- to queue-level egress port schedule in Figure 2.2. The example depicts a port schedule at the flow-level and the corresponding GCL and queue assignments. The flow-level schedule has two contiguous flow reservations assigned to queue 0 and queue 1. The switch port, as depicted, supports best-effort and scheduled traffic. Queues in the switch's egress port are assigned to specific traffic types. In this example, the two highest priority queues,

0 and 1, are reserved for scheduled traffic. When a scheduled traffic flow is scheduled to egress (e.g., t_1), the corresponding queue is opened (queue 0), and all other queues are closed. This effectively disables the strict priority transmission selection, as only one queue is eligible for transmission until the scheduled transmission ends at time t_2. At time t_2, queue 0 is closed, queue 1 remains closed, and the remaining best-effort (BE) queues are opened. The process continues until the port cycle repeats. Note, that we are assuming implicit guard bands, protecting the higher priority scheduled traffic windows for interference from lower priority frames. With implicit guard bands, frames are not transmitted if they cannot be *completely* transmitted within the remaining time of the transmission window.

With scheduled traffic supported by GCLs, multiple queues can be utilized to separate frames of different traffic types or separate frames of the same type. The utilization of multiple queues within scheduled traffic, for instance, allows for a spatial separation of flows, referred to as *flow isolation* [18] or deterministic queues [80]. Frames of different flows need to be isolated to avoid side effects in the event that a flow suddenly goes missing or multiple flows arrive at the same time on different links. Flow isolation can also be achieved by avoiding the dual use of a queue with multiple flows queued at the same time.

Asynchronous: Credit-based Shaper

The Credit-based Shaper (CBS) [42, p. 220] is a shaping algorithm that essentially limits the bandwidth of a single queue to a fraction of the available line rate. This fraction can be controlled by a single parameter, denoted **`idleSlope`**, for the selected CBS queue. The Credit-based Shaper utilizes a credit mechanism to control the queue's frame transmission eligibility. Frames are eligible for transmission if the accumulated credit value is $\geqslant 0$. See Figure 2.3 for an example depiction of the credit value behavior. Credit is accumulated at a rate of **`idleSlope`** in bits per second, starting from zero, when queued frames are waiting for transmission. Note, credit accumulation is paused when a present GCL closes the gate for the corresponding CBS queue. The credit is decreased by the **`sendSlope`** rate, during active frame transmissions of the queue. An empty queue resets the credit value to zero.

The bandwidth limiting behavior of the credit-mechanism ensures lower priorities (queues that are selected later in strict priority) are able to transmit frames in a predictable way. This increases the fairness between traffic classes compared to strict priority transmission selection, by limiting the frame bustiness [79]. Multiple approaches exist to model the delay behavior of the Credit-based Shaper, e.g., [20, 64, 67, 68, 70, 73, 98, 99]. A majority of these approaches utilize the network calculus framework [19, 24, 60] to calculate worst-case delay bounds for CBS traffic classes.

Due to their function and asynchronous nature, shapers like CBS, or Asynchronous Traffic Shaping (ATS) [43, 91] can only give poor jitter (frame delay variation) guarantees. However, works exist that can reduce jitter by damping the traffic rate to achieve more consistent per-hop latency [34].

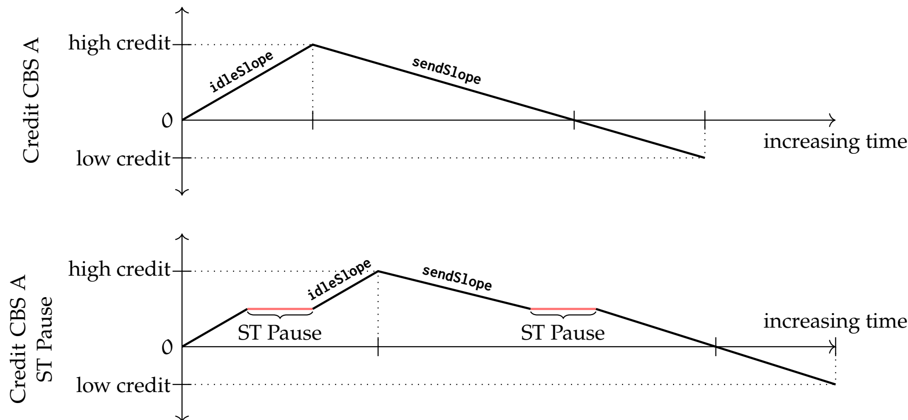


Figure 2.3: Example illustration of the credit behavior of the credit-based shaper for class A. The credit value increases when the queue is not transmitting frames. Frames are eligible for transmission if the credit is not negative. Credit is accumulated at the **idleSlope** rate, and decreases with a rate of **sendSlope**.

2.3 SCHEDULING

Scheduling is an important problem in computer science, and other domains. There is a distinction in scheduling terminology between a *scheduling policy* (e. g., [8, 54, 63, 82]), and a *sequence* or *schedule* (e. g., [18, 28, 90]) of tasks [74]. This distinction is important, as both concepts are applied in TSN mechanisms. Policies are applied for queue or flow selection algorithms, and schedules can describe exact *time*-sequences of relevant Ethernet frames. Scheduled sequences enable the deterministic usage of resources. However, the scheduling problem is in general *NP-hard* [74].

In networking scenarios, to avoid single flows being able to easily influence the performance of other flows, scheduling policies were introduced to improve fairness [92]. Due to the hardware limit of available FIFO-queues within network switches, these scheduling policies usually don't operate on a per-flow level when utilized within switches. This would require one separate queue per flow. Scheduling policies applicable to TSN transmission selection algorithms include strict priority and round-robin varieties.

- The **strict priority** policy selects the highest priority queue that has frames ready for transmission.
- The **round robin** policy cyclically selects queues that have waiting frames. Upon reaching the last queue, it cycles back to the first. Several variants of round-robin exist, such as the Weighted Round Robin (WRR), where the number of transmission opportunities for a queue is influenced by its weight [50], and the Deficit Round Robin (DRR), which accounts for varying frame sizes [88] to enable better fairness. Additional variants are discussed in references such as [5] and [82].

In addition to queueing policies aimed at enhancing overall fairness, some scheduling policies incorporate deadline considerations. These can also be applied in the con-

text of network forwarding. Notably, Rate Monotonic scheduling [63] assigns priorities based on the period of network flows, with those having shorter periods receiving higher priority. Dynamic policies, such as the Earliest Deadline First (EDF) policy prioritize flows according to their upcoming deadlines [60]. Additionally, dynamic variants like Least Slack Time First (LSTF) [62] consider the remaining slack time, defined as the difference between the remaining time until a flows's deadline and the remaining processing time.

Implementing these flow-level scheduling policies in hardware devices presents challenges, as they necessitate either an isolated queue for each flow or arbitrary insertion capabilities to correctly select the queued flows according to the scheduler's criteria.

The deployment of scheduled traffic cannot be implemented using these scheduling policies. Therefore, the focus of the remainder of this thesis is on the schedule sequencing problem.

Scheduling TSN Scheduled Traffic

In classical sequence scheduling problems, constraint-based approaches can be employed to identify viable *optimal* schedules [74]. Optimal, in the sense that a solution is found if one exists. Similarly, TSN-schedulers that utilize the same concept exist. For example, [18] and [83] employ satisfiability modulo theories (SMT)-based formulations to schedule flows using the tool Z3 [97] for use with TSN GCLs and full flow isolation. A constraint-based TSN scheduler for scheduled traffic must define mathematical relationships concerning the flow requirements that are scheduled. This encompasses scheduled traffic parameters like frame sizes and deadlines, but also dependencies among flows, particularly at shared ports, or hardware restrictions like the number of available queues. The implementation of a TSN SMT scheduler used for evaluation purposes is documented in Appendix A.1.1.

However, because constraint-based approaches, depending on the number of flows, can sometimes require hours to days to find viable results [18, 28], efforts are being made to try to improve scheduling speed. The work of [9] enhances the constraint-based solver with a heuristic to accelerate the scheduling speed. Meanwhile, the authors of [36] combine constraint-based scheduling with heuristics by utilizing a bin-packing technique with a first-fit heuristic and scheduling the remaining resources using SMT.

Some TSN flows might require the scheduled sequence to incorporate multicast transmissions. That is, flows that have multiple listeners, where frames get replicated as necessary along their path. In this thesis, we do not explicitly consider multi-path flows. Multicast scheduling requires special consideration within the scheduler [61].

Incremental Scheduling

In dynamic scenarios, when application requirements and network conditions are constantly changing, resetting scheduling results may be infeasible. Re-scheduling may re-

sult in different time slot allocations for a flow, which would require the interruption of the affected application. Therefore, incremental approaches, that build on previous results to adapt the schedule for new requirements, are important. This typically involves fixing the current flow requirements and incorporating new flow(s) and their requirements by modifying the existing schedule. Incremental approaches that also utilize constraint-based methods exist. Pure scheduling heuristics also typically work in a "keep-modify-integrate" manner, instead of combining flow admissions together.

The incremental approach is relevant to this thesis, due the need to adapt configurations at runtime. Several contributions are present in the literature. In the following we highlight some approaches:

In [71], the authors introduce an incremental scheduling method for networks, specifically limiting scheduling operations to end-nodes within their framework denoted Time-sensitive Software-Defined Networking (TSSDN). This approach reduces the need for scheduled switch nodes, although it is constrained by the absence of intermediate scheduling capabilities. The approach employs an integer linear programming (ILP) strategy to formulate the schedule configuration.

A method for rapid re-scheduling in the event of network node failures is given by [25]. The approach introduces SMT constraints to reassign routing of faulty nodes to a backup route. In scenarios where this node-limited re-scheduling is infeasible, the algorithm employs a recursive backtracking technique, revisiting and adjusting the schedules of upstream nodes until a feasible scheduling solution is found.

Pure heuristics

In the literature, pure scheduled traffic heuristic schedulers operate in an incremental manner. A heuristic approach to use multiple queues for flow isolation and also using relaxed jitter requirements is given in [17]. Whereas the scheduling heuristic, given by [6], enables a task-based abstraction scheduling, using a ordering function to select the next incrementally added flow.

The concept of a "soon-as-possible" strategy, we refer to as first-fit heuristic scheduling, is presented in [75]. This methodology aims to schedule flows at the earliest possible opportunity.

Additionally, an alternative strategy proposed by [33] advocates for the shared utilization of GCLs windows to enhance the dynamic capabilities of the implemented schedules. Although this approach compromises flow isolation and elevates jitter levels, it has the potential to achieve greater dynamicity, provided that the flow requirements can accommodate the reduced guarantees.

Wait-free vs Queueable

The authors of [61] present a scheduling heuristic that reduces the search space to *no-wait* flow schedules while permitting non-zero jitter. Notably, this approach also can enable multicast transmission, thus broadening its applicability in complex network scenarios. A flow schedule designated as *no-wait* ensures that a flow is scheduled with minimal latency during the transmission of its frames within the network. This

effectively eliminates queueing delay, as frames are immediately forwarded to their subsequent destinations. In [13] the authors also employ a no-wait strategy, both with random and greedy scheduling approaches, with the latter incrementally scheduling a flow by minimizing contention between existing flows.

Similarly, the authors of [22] employ no-wait scheduling with means of an ILP formulation. They also employ a first-fit heuristic that adheres to no-wait flows and is capable of incrementally adding flows. It is important to note that, in general, first-fit heuristics may allow the introduction of queueing delays. As a consequence, they do not limit the search space exclusively to scenarios without no-wait constraints.

In the concepts described in this thesis, we consistently assume that all flows are permitted to accumulate queuing delays along their paths within the network.

Joint Routing and Scheduling

Few approaches integrate routing into the primary scheduling process. Incorporating routing significantly increases the complexity of problem formulations, rendering them challenging to manage due to their expanded size. Thus, approaches use well-known metrics, like shortest path routing, as a route selection mechanism, or do not specify how the route is obtained at all.

Approaches that explicitly consider the path selection typically employ custom metrics to sort paths for their scheduling viability. The approach [72] introduces the *flowspan* metric, where the goal is to keep it minimal. The flowspan is the total time, from the first transmission to the end of the last transmission, that all scheduled flows require in the network. Whereas in [61], the proposed heuristics continuously use only the shortest paths available and switch to a fallback if paths are ineligible for scheduling.

A noteworthy contribution is [12], which builds upon the work of [31]. This approach is a strategy for scheduling TSN scheduled traffic while simultaneously accommodating the requirements of another TSN mechanism, specifically, the Credit-based Shaper. Our approach of mechanism combination, described in Section 6.3, primarily serves as an extension of our flexibility metric, additionally enabling limited scheduling capabilities. Further, [12] introduces a sorting metric that also selects potential routes prior to scheduling. This metric incorporates scheduled traffic utilization and the number of CBS flows.

In summary, various approaches can effectively schedule scheduled traffic flows. Among these, heuristic methods balance overall schedulability with rapid generation and incremental efficiency. However, to the best of our knowledge, all existing methods demonstrate a limited understanding of TSN schedule flexibility. Only a few approaches incorporate metrics to further steer their scheduling decisions for improvements, such as the overall schedule utilization, with means of flowspan [72] or path utilization [12].

2.4 SOFTWARE-DEFINED NETWORKING

With the advent of Software-defined Networking (SDN), the paradigm of splitting the control- and data-plane of network forwarding devices was established. The control-plane is responsible for deciding how packets are being forwarded, and the data-plane forwards packets according to the deployed configuration received from the control-plane. Interfaces to and from the logically centralized control-plane (controller) are split into north- and south-bound Application Programming Interfaces (APIs) to configure the network. The north-bound API is used by users of the network to interact with the controller, while the south-bound API enables the configuration of network elements [52]. Refer to Figure 2.4 for a schematic overview of this split control- and data-plane paradigm of SDN. One prominent example of a south-bound API is *OpenFlow* [69]. Devices providing the OpenFlow API allow for the configuration of the device's flow tables. Flow table entries are matched against incoming frames and allow the execution of specific actions, such as frame forwarding, modification, or dropping. While OpenFlow provides a unified API for configuration, it is still limited. The forwarding behavior itself cannot be controlled. The domain-specific language *P4* [14], aims to narrow this gap by allowing for more flexible control of the data-plane forwarding behavior. With a C-like syntax, P4 allows for the support of custom protocols.

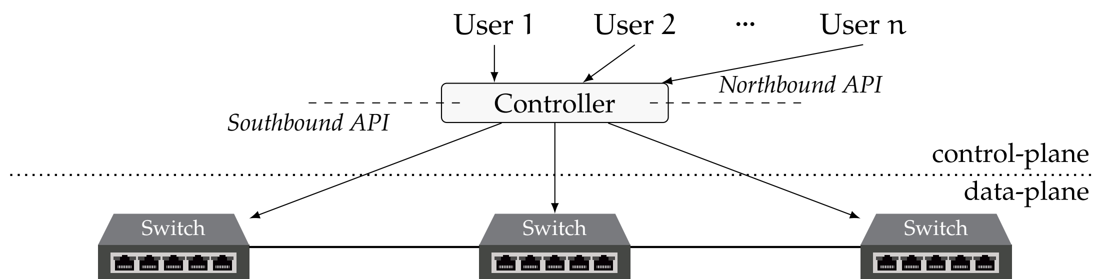


Figure 2.4: SDN assumes a split control- and data-plane. Users access the centralized controller via the northbound API. The controller configures the network devices via south-bound interfaces.

P4 code adheres to a directed acyclic graph structure, prohibiting loops or backward jumps to preceding statements. This allows P4 code to be executed in a pipelined fashion. Products exist based on application-specific integrated circuits, such as the Intel Tofino platform [2, 48], that support line-rate packet processing and forwarding under these P4 constraints. Programmers can define custom parsing and packet processing behaviors. Packets are manipulated using a set of instructions (*actions* in P4) and lookup operations (*match-action tables* in P4). For instance, P4 enables modifications such as rewriting header fields, and adding or dropping headers. Stateful operations, such as registers, depend on available architecture-specific API definitions. The mathematical operations in P4 are also constrained; notably, P4 lacks inherent support for division. The application of P4 has been demonstrated to enable an offloading of specialized tasks to the data-plane. For example, the forwarding decisions of publish/-

subscribe brokers can be implemented through P4 programs [55, 94]. Moreover, P4 can accelerate the indication of the explicit congestion notification (ECN) header field [56] in TCP traffic and support efficient load balancing strategies [15, 51].

TSN, while not directly associated with SDN, still allows the matching of SDN concepts to TSN specifications when used with TSN's fully centralized configuration model. For instance, we can access the configuration of TSN devices with a south-bound API. We further elaborate on how this impacted design decisions for this thesis in Chapter 3.

While P4 significantly enhances the capabilities of forwarding devices, it is not a substitute for TSN mechanisms. For example, the programmable forwarding hardware supported by P4 cannot manipulate the scheduling logic required by certain TSN mechanisms. The *traffic manager* within P4 switches [58], responsible for queuing and scheduling, is not programmable via P4. It is positioned between the ingress and egress pipeline processing stages of P4. The ingress pipeline's processing capabilities are limited to configuring exposed API parameters set by the architecture, which in turn define the traffic manager's behavior. A typical operation involves selecting the appropriate egress port for the packet currently being processed. Subsequently, the egress pipeline processes the packet further post-scheduling by the traffic manager.

Hence, when only using P4, we are limited to approaches that do not directly control the scheduling logic of the traffic manager (cf. Section 6.2). An approach to allow for programmable schedulers is provided by push-in-first-out (PIFO) [89] queues. PIFO suggests the usage of hierarchical priority queues, i.e., queues in which the insertion order can be specified according to a ranking algorithm. However, PIFO-enabled devices are not currently widely available. This limitation restricts the emulation of actual TSN mechanisms on typical SDN-hardware (cf. Section 6.1).

2.5 FLEXIBILITY

Flexibility can be defined in various ways. For instance, flexibility in manufacturing systems, as surveyed by [86], is divided into multiple distinct *types* of flexibility. Among them are *machine flexibility*, *operation flexibility*, *process flexibility*, and many others.

Within the surveyed approaches, the consideration of speed, number of possible actions, and costs is common in their conceptualization of flexibility. In the domain of business processes, the survey by [84] groups flexibility types according to flexibility strategy: *by design*, *by (temporary) deviation*, *by underspecification*, and *by change*. This notion of flexibility is assumed to be applicable at runtime. As this thesis is mainly concerned with sequencing flexibility in the context of TSN scheduled traffic, the flexibility type is akin to the *operation flexibility* utilizing a *by change* strategy. Operation flexibility refers to the ability to achieve the result (e. g., a part production) in different ways. Utilizing other strategies could lead to either intentional over-provisioning, which is costly, or to unnecessary reservations, which degrade the performance of currently deployed traffic flows.

A possible quantification for operation flexibility, for instance, is given by [81]. The *sequencing flexibility measure* considers possible operation sequences and the overall number of operations for a task:

$$\text{Sequencing flexibility measure} = 1 - \frac{2\Lambda_i}{n_i(n_i - 1)}$$

With Λ_i being the number of transitive precedence arcs in a graph of operations for task i, and with n_i being the total number of operations for task i. Hence, it measures the flexibility on a per-task basis. Conversely, the flexibility metric (cf. Chapter 4) of this thesis adopts a path-based perspective. This emphasizes the flexibility quantification of entire scheduled paths rather than distinct flows. This path-based approach reflects the stringent routing requirements of scheduled traffic, allowing system-wide adjustments and decisions rather than decisions for individual flows.

Network Flexibility

The term *network flexibility* is readily associated with the concept of SDN, as the centralized controller can easily adapt the forwarding elements through the south-bound API [49]. However, the means of deploying changes, does not inherently provide the network's flexibility. For instance, wireless networks are considered to be more flexible than their wired counterparts [87]. SDN can help improve the network flexibility by adding configuration paths or allowing for faster configuration adaptions. The possible network adaptations are, however, limited by the deployed applications.

The generalized model to measure network flexibility by [7], defines network flexibility in the following way: "Given the demands the communication network has to respond, network flexibility is the ability of the network to adapt its state to satisfy the new demands promptly and with little effort" (p. 15).

The consensus on the conceptualization of flexibility, derived from some understandings [7, 30, 84, 86], suggests that it is captured by metrics of speed, possible actions, and associated costs. Similar to the previously mentioned *sequencing flexibility measure*, the model of [7] quantifies flexibility by electing to find achievable changes for the application demand, however, that also depend on the time and cost of introducing these changes:

$$\mu(\mathcal{A}_X(T, \mathbf{C}))$$

With a set of achievable changes $\mathcal{A}_X(T, \mathbf{C})$ for network configuration X under time T and cost $\mathbf{C}$ requirements, and with the measure μ, which can represent the count of achievable changes. This framework is highly versatile and can be adapted to various scenarios and contexts. Additionally, the authors suggest a potential normalization using the ratio $\frac{\mu(\mathcal{A})}{\mu(\mathcal{A}^*)}$, where $\mathcal{A}^*$ denotes the maximal number of changes across all possible network configurations. However, the model does not inherently provide a method for identifying the set of achievable changes in particular contexts. Specifically, in the context of TSN, this thesis outlines an approach to evaluate the achievable changes for

isochronous traffic. It is important to note that the costs and timing of changes are not specified within the scope of this thesis.

We adapt the definition of flexibility for TSN in this thesis, and reiterate the definition of flexibility from the first chapter:

> **Definition** — *Flexibility* is the ability to accommodate future changes of configurations at runtime.

In the literature, several metrics capture the behavior of scheduled traffic to varying extents. For instance, the term "makespan" refers to the total time required for the completion of all tasks within a system [74]. Similarly, the term *flowspan*, adapted from the makespan concept, is introduced by [72]. Flowspan denotes the total time required to handle all scheduled flows, distinct from both the hyperperiod and individual flow periods.

Another metric is the route ordering metric proposed by Berisa et al. [12]. This metric constitutes a weighted sum with weights $w_1, \ldots, w_4$ for path P and link p:

$$w_1 \cdot n_P + w_2 \cdot \overline{u}_P + w_3 \cdot \hat{u}_P + w_4 \cdot n^p_{AVB}$$

Here, n_P represents the normalized number of links between the talker and listener, $\overline{u}_P$ denotes the average schedule utilization along the path P, and $\hat{u}_P$ is the standard deviation of the path's schedule utilization. Notably, this metric integrates CBS flows, where n^p_{AVB} represents the normalized number of CBS flows at link p, relative to the total number of CBS flows in the network.

In contrast to our approach, which is described in detail in Chapter 4, both metrics primarily focus on utilization. Unlike these metrics, ours adapts according to the actual flow requirements. The Berisa metric, like ours, is also a path-based metric but restricts the view to specific ports for CBS traffic.

2.6 transitions in time-sensitive networking

In communication systems, transitions between similar mechanisms can increase the overall system's flexibility [4]. For instance, a mobile phone might transition from a cellular network to a WiFi connection to maintain the connection at a stable quality level.

Similarly, in the context of Time-Sensitive Networking, transitions between network configurations are crucial for accommodating the dynamic requirements of industrial applications or those applications that are constantly changing. TSN, by design, provides mechanisms for deterministic data delivery. This is in contrast to networking mechanisms where this inherent guarantee cannot be given and must be achieved through a mechanism transition.

The concept of transitions within TSN thus extends beyond mechanism transitions. We need to enable the network's configuration to support changes of application requirements. This thesis proposes the usage of a centralized control design, akin to SDN, to facilitate this increased flexibility. With centralized control, we can not only facilitate a dynamic adaptation of the network configuration but also ensure that the configuration transitions are seamless. Seamless transitions are facilitated by utilizing the flexibility metric for scheduled traffic described in this thesis.

3 FLEXIBILITY-BASED TSN MANAGEMENT

In this chapter, we describe the design of a flexibility-based Time-Sensitive Networking (TSN) management scenario. This sets the stage for the subsequent chapters, where we implement and discuss approaches to achieve the research goals outlined in Chapter 1. The scenario described in this chapter is reflective of real-world situations in TSN network management and operation.

3.1 MANAGING SCENARIO

Connecting the digital and physical worlds to create more efficient systems in industrial processes is a current trend in automation and manufacturing processes. The term *Industry 4.0* is often referred to when referencing, among other advances, the desired increase in interoperability, flexibility and connectivity within industrial processes [59, 77, 95]. More specifically, in a factory context, the term *smart factory* and Industrial Internet of Things (IIoT) was coined to describe a factory with machines that are connected using traditional packet-based networks [37, 53, 66]. TSN lends itself as a key enabling technology for smart factories to support the traffic requirements of industrial automation applications [10, 66].

Industrial applications have multiple different types of real-time traffic requirements, depending on the specific application. Different traffic types might also be used simultaneously. Further, TSN communication can be used to communicate within or between machines. E. g., a programmable logic controller (PLC) of a conveyor belt communicates with a PLC of a robot. Below are some noteworthy traffic types of industrial applications as specified by [10] and [1]:

- **Best-effort (BE) traffic:** Traffic with no service requirements. It is transferred as effectively as the network conditions allow. Packets of this traffic type may also be dropped if necessary.
- **Audio/Video:** Periodic traffic with guaranteed latency and bandwidth. The data size is bounded.
- **Network control:** Sporadic traffic with strict priority. The network control traffic type is not available for normal network end-devices.
- **Cyclic real-time:** Periodic traffic with guaranteed delivery deadline and bandwidth. Periods are multiples of the network cycle time. The data size is bounded to a single packet. Network access is based on local time.
- **Isochronous cyclic real-time:** Periodic traffic with cycles ranging from $1\mu s$ to $4ms$, guaranteed delivery, deadline and bandwidth. The data size is bounded to a single packet. Network access is based on synchronized time.

With TSN flexibility as the main research subject of this thesis, the traffic type of primary relevance is the isochronous cyclic real-time traffic type. This traffic type promises strict real-time guarantees with short periods. The requirements of this type can be satisfied with gate control lists (GCLs) and scheduled traffic (ST). BE-traffic is still assumed to be present at all times in the network, and its presence or absence must not interfere with any given service guarantees.

Network Management

With isochronous traffic, we assume a time-aware talker (the network application) is participating in the network. How we assume the network to be managed is similar to the Software-defined Networking paradigm using a logically centralized controller. The amendment IEEE Std 802.1Qcc [44] defines a fully centralized TSN configuration model as standard functionality. We chose to be compliant to the standard, as the fully distributed configuration model is not intended to deploy ST. Additionally, [44] recognizes significant advantages in the configuration computation when a fully centralized model is used. A centralized configuration model reduces complexity significantly when network wide schedules and knowledge of application requirements is needed.

The central controller is divided into two logical entities: Centralized User Configuration (CUC) and Centralized Network Configuration (CNC). The CUC serves as the logical entity to represent all network applications to the CNC. For this, the CUC needs to collect their requirements and configure the TSN specific features. The CNC serves as the central network configurator. The CNC interfaces with the CUC to exchange traffic requirements and network configurations. The specifics of the CUC-CNC protocol are being considered in IEEE Std 802.1Qdj [38]. However, the protocol between CUC and actual applications is not standardized.

We provide a high-level overview of this paradigm in Figure 3.1, depicting a small network that includes three TSN switches, two PLCs, the controlled application, and a combined CUC/CNC entity. The network features two flows: f_1, a control-to-control flow, and f_2, which controls the application. It is important to note that this configuration is not static; PLCs or applications may be repositioned or request changes in flow requirements through interactions with the central controller.

The CUC does not autonomously discover end-stations and applications in our scenario. Instead, it receives requests for new flow requirements when application requirements change or emerge. It also might choose to aggregate multiple requests before contacting the CNC. Any participating device can initiate the request, but typically, a managing device of the relevant application is responsible for this task. Communication with the CUC can occur off-network or using the deployed TSN infrastructure. When the TSN network is used, the CUC needs to be reachable via a non-management port of a TSN switch. In IEEE Std 802.1Q [42, p. 1673], the flow specification for talkers/listeners encompasses a variety of use cases, leading to a verbose hierarchy. To simplify, we have condensed the *traffic*, *end-station*, and *network requirements* of flows to these six essential elements, suitable to describe the requirements of a flow for isochronous real-time traffic: **`Flow ID`**, **`Source`**, **`Destination`**, **`Frame interval`**,

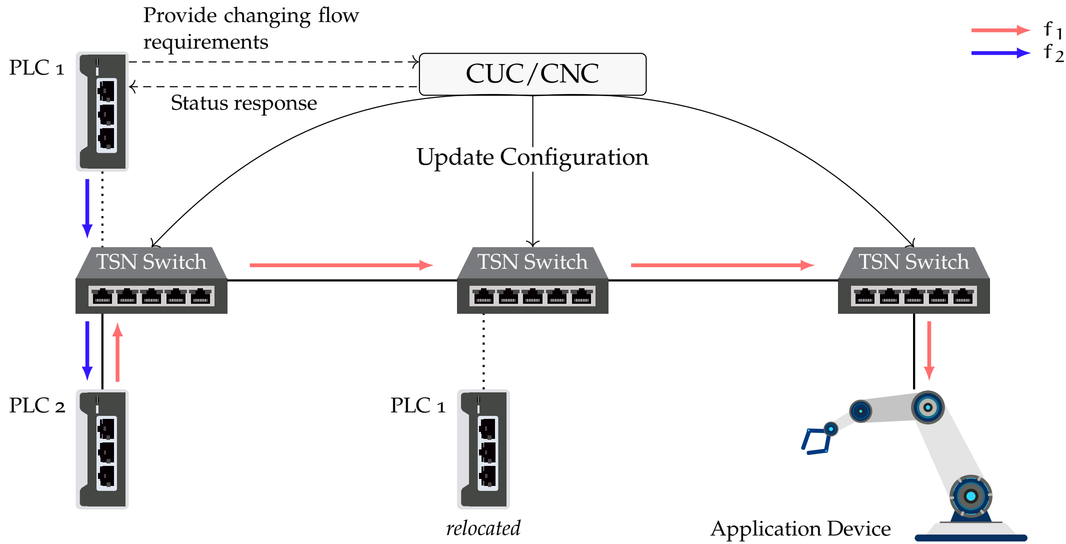


Figure 3.1: The generic centralized configuration model assumed throughout this thesis. A central controller manages the TSN network, consisting solely of TSN-capable devices. Network applications, e. g., PLC request their flow requirements at the CUC. Figure derived from [26, 28].

`Frame size` and **`Deadline`**. A description of each flow parameter is given in Table 3.1. We assume these parameters as specification for a flow's requirements throughout the remainder of the thesis.

On a high-level, the request/reply protocol for applications to interact with the CUC is assumed as follows:

1. Applications request flow admissions to the network by sending change requests to the CUC. Requests include new or updated flow parameters (cf. Table 3.1). Multiple requests can be made non-blocking. Applications should also inform the CUC when flows become obsolete.

2. Applications await a status response from the CUC.

3. The Status response is used to confirm a successful flow admission within the network configuration. The TSN specific features of the talkers, namely the priority code for correct queue assignment, timed sending and clock synchronization, can then be configured.

Like the flow requirement specification, IEEE Std 802.1Q specifies flow status [42, p. 1686]. Due to the consideration of multiple use cases and failure scenarios, this specification is similarly verbose. Hence, we reduce the flow status to its essential parameters. We assume the CUC includes the parameters **`Is admissible`**, **`PCP`** and **`Allowed starting time`** in the status response to the applications for each requested flow. The **`PCP`** reflects the priority identifier to correctly separate traffic for flow isolation purposes.

Table 3.1: Description of flow parameters for isochronous traffic.

Parameter	Description
Flow ID	A unique identifier for the flow, determined in conjunction with the source.
Source	The interface identifier from which the flow is initiated.
Destination	The target interface identifier where the flow is directed.
Frame Interval	The regular time interval at which frames in the flow are transmitted.
Frame Size	The maximum size of each data frame within the flow.
Deadline	The maximum allowable end-to-end latency for a frame to meet its timing constraints.

If a flow admission request results in an admissible flow, i. e., succeeds, the CNC already initiated the required configuration changes within each affected switch. Required changes include deploying replication and forwarding rules and adjusting the affected TSN parameters. Note, there is no priority for isochronous flows, meaning all requested flows are of equal priority to the network. We also assume there is no flow request priority, i. e., flows are not evicted without explicit application request, e. g., in order to clear necessary capacity for new flows with higher application priority.

3.1.1 *Network Model*

We model a given network as a graph $G(V, E)$, where nodes V represent time-aware devices such as switches, talkers, and listeners, and edges E represent directional links connecting these devices. Let v_i and v_j be two distinct nodes in V. These nodes are connected if there exists an edge $p \in E$ such that $p = (v_i, v_j)$, denoting the physical link and uniquely identifying source and destination ports between v_i and v_j. In duplex networks the reverse direction $p = (v_j, v_i)$ exists as well. A path P of m-hops in the network is specified by a sequence of directed links $P = (p_1, \ldots, p_m)$, called ports.

The CNC tries to admit the set of isochronous flows $F = \{f_1, f_2, \ldots\}$ to be simultaneously deployed in the network. A flow $f_\delta \in F$ has traffic parameters that are specified by the application (cf. Table 3.1). The set of flow-level port schedules is given by $S = \{s_p \mid p \in E\}$. Each schedule s_p is a sequence departure times $s_p = (s_p^1, s_p^2, \ldots, s_p^{X_p})$ for the given source port p. We define a mapping function $M : S \to F$ such that $M(s_p^i) = f_\delta$, where s_p^i is the departure time of the i-th schedule entry, and f_δ is the flow to which this entry belongs to. This mapping ensures that each departure time is associated with a specific flow.

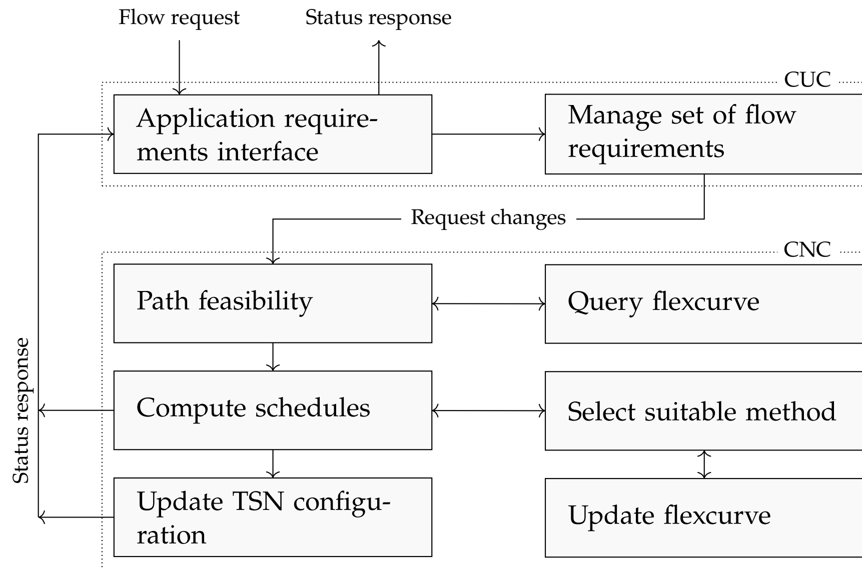


Figure 3.2: High-level control flow of CUC and CNC. The control flow also depicts the hooks of research goals 1 and 2. Namely, the consultation of the flexibility metric (flexcurve) to decide the feasibility of requested changes and steer the scheduling decisions. Figure derived from [26, 28]

3.2 CONTROLLER

All approaches and their prototype implementations (cf. Chapters 4 to 6) semantically adopt the centralized CUC/CNC architecture with the protocol as mentioned above for applications to request their flow requirements. Applications communicate directly with the CUC, send requests, and receive status responses. In our demonstration [27] to showcase TSN management with flexibility awareness, the CUC/CNC entities are combined on a single host. Specifics about the prototype implementation of this flexibility-aware controller are described in Appendix A.1.

In this thesis, we can extend the capabilities of the CUC by integrating approaches from Research Goal 1 and 2 to improve the achieved flexibility. Figure 3.2 depicts a possible control flow of this extended CUC/CNC capability. The control flow begins with a flow request from an application to the CUC. The CUC provides an Application Programming Interface (API) for applications to access. After the CUC receives a flow request, it updates its internal set of requested and active flows. The CUC can delay further processing for aggregation possibilities or immediately request changes from the CNC. After the CNC receives the requested changes from the CUC, it checks first for feasibility. Feasibility is checked by consulting the flexibility metric (cf. Chapter 4, flexcurve). The flexcurve enables a path selection based on the level of path flexibility if multiple paths are available between the talker and listener. It can also approximate whether an admission on the selected path is possible. After path selection, the CNC needs to create the updated TSN configuration. Depending on which changes are requested by the CUC, the CNC needs to choose a suitable method to satisfy the change

request. Methods can include heuristics to allow for partial reconfigurations and can also incorporate the flexcurve to steer the resulting configuration (cf. Chapter 5). Some requests, which incremental methods cannot satisfy, might require complete reconfigurations or are otherwise not schedulable due to insufficient capacity or unfit topology. In this case, the change is declined, and the field **Is admissible** is set to **False**. If a configuration can be changed without affecting deployed applications, the CUC initiates updates on affected network devices. The status response confirms success with **Is admissible** set to **True**. The field **PCP** is set to the assigned queue identifier, and **Allowed starting time** is set to the earliest absolute network time when affected devices can accept the changed traffic. This status response is given for each affected flow.

For scheduled traffic, the **Allowed starting time** is indirectly given by the *List Config state machine*, specified in IEEE Std 802.1Q [42, p. 228]. The List Config state machine exposes the variable **ConfigChangeTime**, which specifies the time the new configuration is active at the affected port. The **Allowed starting time**, relevant to the time aware talker, is determined by adding the flow's *schedule offset* at the talking port to **ConfigChangeTime**. The scheduling procedure yields flow schedule offsets, indicating the relative time offset from the start of the cycle period when flows are scheduled. In the case, the **ConfigChangeTime** differs at different ports, the CNC needs to ensure consistency and aligned GCL cycles. The CNC can achieve consistency by moving the cycle base time to a future time. This results in the **ConfigChangeTime** being set to a CNC specified time. Inconsistent **ConfigChangeTime** variables can occur when different cycle periods are deployed.

Hardware Abstraction Layer

When the TSN configuration changes, it must be deployed. The CNC initiates updates of affected devices after successfully concluding the scheduling process based on the requested changes from the CUC. The method to compute the requested changes can operate on different abstraction levels. For instance, methods operating solely on a flow level cannot be directly deployed using the GCL mechanism. The reason is that TSN GCLs affect the data transmission per queue, not per flow. Therefore, to deploy flow-level schedules, additional information is required.

A possible solution is to use a hardware abstraction layer (HAL), that sits below the scheduling logic of the CNC. A possible HAL layering is depicted in Figure 3.3. It is split in north- and southbound API. The northbound API interacts with the CUC. Here, the CNC has access to various methods to create an updated configuration model to satisfy the flow request. Methods can include classical constraint-based schedulers, heuristics and the flexcurve. Whereas the southbound API interacts with hardware devices for deploying the configuration. Different hardware devices can have different capabilities. For instance, if a device supports GCLs, the HAL populates the necessary list entries to deploy the configuration model, and selects the correct queue assignments to guarantee proper flow isolation. Other devices might support different mechanisms, such as push-in-first-out (PIFO) queues that allow direct deployment of

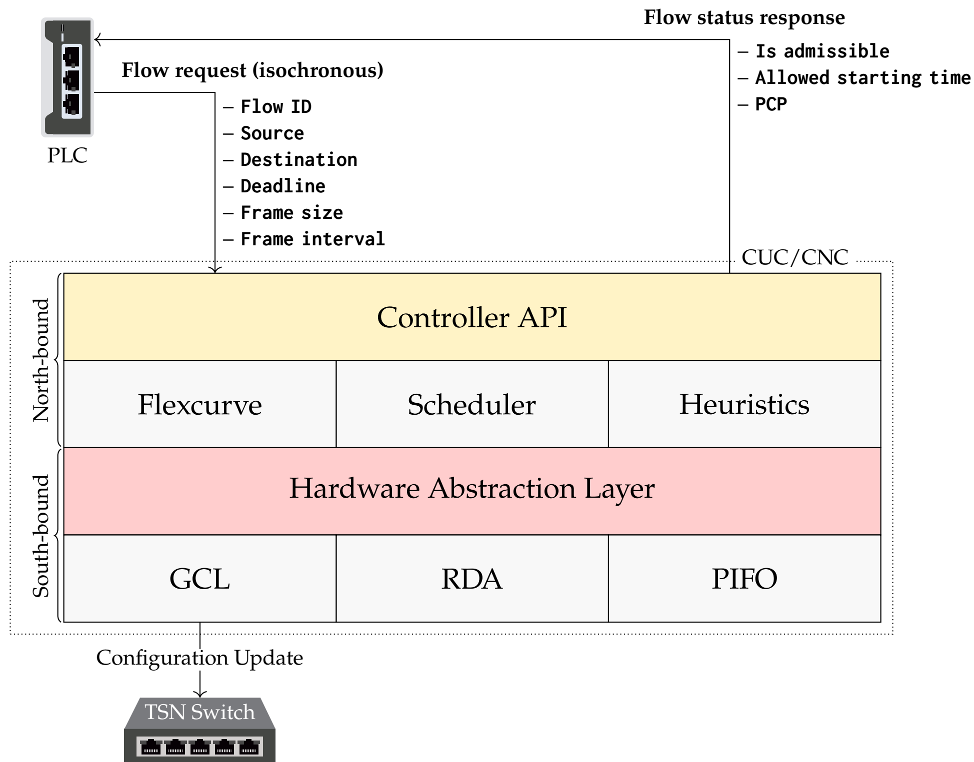


Figure 3.3: A hardware abstraction layer can be used to translate results from the scheduling process to individual device configurations.

flow-level schedules and residence delay aggregation (RDA), which does not support isochronous traffic.

A possible translation process of flow-level schedules to GCLs is discussed in Section 5.3. The deployment procedure of PIFO and RDA is discussed in Sections 6.1 and 6.2 respectively.

4 FLEXIBILITY NOTION

In this chapter, we introduce the flexibility metric *flexcurve* for Time-Sensitive Networking (TSN) to assess the level of flexibility in TSN scheduled traffic. This chapter addresses Research Goal 1. Initially, we present the fundamental concept, followed by the formulation of the *flexcurve* in Section 4.1, and incorporating extensions for deadline-awareness and overall speed enhancements. We conduct an empirical evaluation of the proposed approaches in Section 4.2.

Isochronous real-time traffic requires a careful configuration planning process, which includes flow scheduling mechanisms orchestrated by the Centralized Network Configuration (CNC). The resulting schedules of this planning process are opaque to possibilities for future changes. This is the case for both flow- and queue-level schedules. Without specific indicators, we cannot directly determine which schedule offers a higher level of flexibility between two given schedules.

Although flows can be removed if the scheduling mechanism supports flow isolation, the extent of the resulting gain in flexibility remains uncertain. Similarly, the flexibility of the schedule to accommodate new flows is also unclear. There is inherent uncertainty associated with introducing changes. A metric is needed that can accurately reflect the flexibility. As described in the previous chapter, the scheduling process must be revisited whenever new flows are requested. Depending on the scheduling mechanism, this could necessitate a complete rescheduling, potentially impacting all deployed real-time flows. Furthermore, even if incremental updates are feasible, the admission of new flows is not guaranteed.

To quantify the flexibility, we need to select sensible indicators to specify the schedule flexibility level. Generalized models to measure network flexibility as of [7] are suited to describe the flexibility of a networked system. The flexibility model requires reporting the achievable changes for the application demand, that also depend on the time and cost of introducing these changes to the network. In this chapter, we propose how to assess the achievable changes for isochronous traffic within TSN. It is very hard to completely support such a flexibility model for TSN scheduled traffic, that also includes a cost-model to achieve these changes. Hence, we limit our assessment to important parameters for isochronous traffic flows.

On the other hand, using well-known, simple metrics like utilization and fragmentation are not well suited to reflect TSN scheduled traffic flexibility. They can lead to completely opposing flexibility levels if chosen as flexibility indicators. The fragmentation of a schedule reflects the count of interruptions between reservations. The utilization of the schedule reflects the count of reserved slots. Figure 4.1 illustrates schedule utilization and fragmentation with an example: Both depicted port schedules contain the same number of reserved slots (red), indicated by the same slot utilization of 5. It is not specified which slots are assigned to specific flows. The red highlight merely

indicates if a reservation is made by the scheduler for this time point. The schedule on the left (s_{p_1}) exhibits a relatively high fragmentation with four interruptions, whereas the schedule on the right (s_{p_2}) shows fragmentation that is low, with only one interruption, resulting in one contiguous gap within the cyclic repetitions of the schedule.

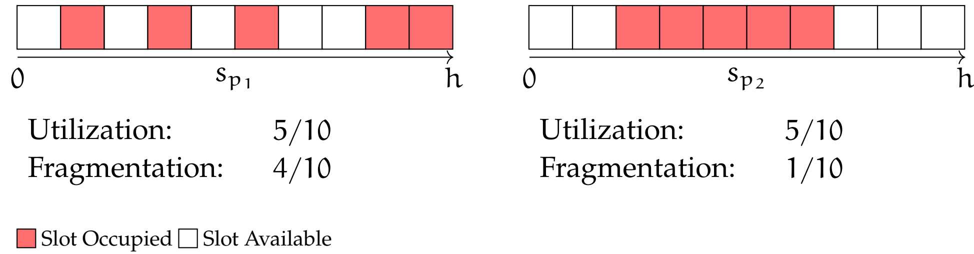


Figure 4.1: Depicted are two port schedules with different slot utilization and fragmentation. The metrics utilization and fragmentation are not well suited to describe possible changes, i. e., flexibility in port schedules. Schedule fragmentation reflects the count of interruptions between occupied slots and utilization gives the overall total count of occupied slots. Both metrics do not regard the timed nature of isochronous traffic.

When the fragmentation metric is used individually, it can only give little information on possible changes for the given schedule. We can only conclude that gaps are available within the hyperperiod h, which have some capacity for future reservations if the number of fragmented reservations exceeds 0. The utilization can give an upper bound of possible admissions until the schedule's capacity is fully exhausted. Therefore, both utilization and fragmentation specify flexibility only to a minimal degree.

Utilization and fragmentation metrics are employed in block-based allocation strategies within data storage scenarios. Although real-time traffic scheduling and block-based allocation may appear similar at first glance, due to both processes involving the placement of data or information, significant differences exist that make the reuse of concepts challenging or unfeasible.

The key distinctions between block-based allocation and the scheduling of real-time flows are as follows:

- **Contiguous Slot Reservations:** Unlike data storage blocks, that can be stored non-sequentially or not contiguous, isochronous flows require slots to be reserved contiguously. This means a frame for an isochronous flow being scheduled at a port cannot be preempted by other traffic at its initially scheduled time and resumed later.

- **Slot Eligibility:** In port schedules, each slot is associated with a specific transmission time, rendering arbitrary time assignments for flows infeasible. For example, scheduling the transmission of frames too late may violate flow requirements, potentially causing deadline misses or frame drops.

- **Synchronization Across Hops:** The scheduler is assuming a time-synchronized network, meaning each port schedule is synchronized in execution. This means

each slot placement depends on the allocation of the previous and subsequent port schedules, to create one functioning global traffic schedule. Multi-hop timing dependencies are not considered in block-based storage systems.

The fact that port schedules are not isolated but depend on the previous and subsequent schedules makes an isolated view less useful. An empty port schedule, able to accommodate a flow with any requirements, is irrelevant if along the flow's path a bottleneck limits this flexibility. In addition, the flexibility of two distinct paths in the network can be vastly different. Therefore, a metric to measure scheduled traffic flexibility should reflect the path and consider all schedule allocations along this path.

4.1 flexcurve: a notion of flexibility for tsn

We propose the flexibility notion *flexcurve* for TSN scheduled traffic. The flexcurve notion quantifies the capacity for accommodating new flows with selected flow requirements along a specified path. The intuition behind this is, that paths with lower flexibility offer limited capacity for new admissions, whereas those with higher flexibility can support more flow requests. Flow exclusions are implicitly accounted for, as the exclusion of a flow increases the potential for new admissions. The modification of flows is represented on the flexcurve in a two-step process: initially as a flow exclusion (i), followed by a flow admission (ii). A flexcurve value of zero should indicate complete inflexibility, meaning no additional flows can be admitted. To quantify the path flexibility, the metric needs to consider the allocation of the schedule and the flow requirements. This goes beyond simple utilization and fragmentation. In the following we assume the network model introduced in Section 3.1.1.

The flexcurve, reflecting schedule path capacity for new flows, necessitates that its simplest form incorporates the flow frame size and path. Thus, a basic flexcurve associated with a path $P = (p_1, \dots, p_m)$ of length m, quantifies the number of feasible flow configurations for a flow characterized by frame size c and path P. This formulation notably excludes the frame deadline and specific frame interval parameters. The network's hyperperiod is denoted by h. Because a frame transmission occurs once within a port schedule the frame interval is assumed to be h time slots for a basic formulation.

To gain insight into a specific schedule's capacity, we denote the cumulative capacity for port schedule s_p up to slot index n with $n \in \{1, \dots, h\}$ by

$$C_p(n) = \sum_{\beta} 1_{\{n \geqslant \mathcal{T}_{p,\beta}\}} \tag{4.1}$$

where $1_{\{\cdot\}}$ maps to the value 1 if the argument is true, and 0 otherwise. The time point of the β-th free slot at $\vec{s}_p$ is given by $\mathcal{T}_{p,\beta}$.

The gap duration, i. e., number of contiguous free slots, after the last reservation ends at a port schedule s_p up to index h in s_p is given by φ_p. We define the end-shifted sequence of s_p as $\vec{s}_p = (s_p^1 + \varphi_p, s_p^2 + \varphi_p, \dots, s_p^{\chi_p} + \varphi_p)$, with χ_p being the number of departure times. Figure 4.2 depicts an example of the cumulative capacity with s_p and $\vec{s}_p$ as schedule basis for $\mathcal{T}_{p,\beta}$.

Note, we use the end-shifted sequence $\vec{s}_p$ instead of s_p to consider the cyclic nature of the schedule (cf. Lemma 4.2). The example depicts a port schedule s_p with gaps near the cycle boundary h, whereas the end-shifted port schedule $\vec{s}_p$ is shifted for the duration of the last gap $\varphi_p = 2$. The corresponding cumulative capacity $C_p(n)$ reaches the maximum value earlier than with s_p as data basis. However, the overall utilization did not change, hence the cumulative capacity for both depicted port schedule variants eventually reaches the same value of 5.

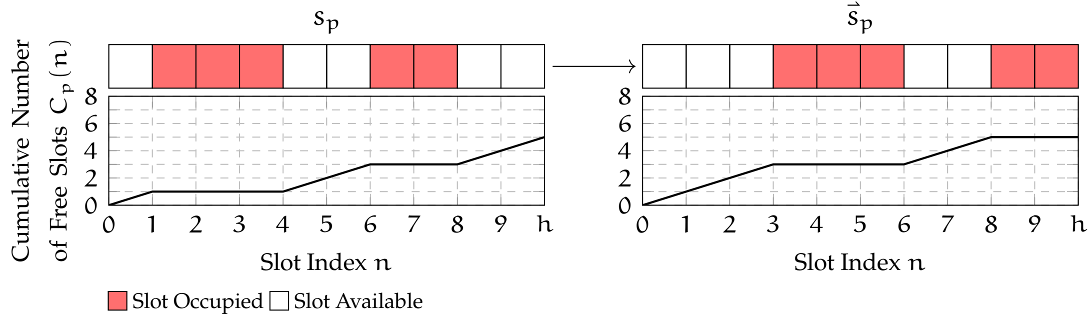


Figure 4.2: Example: A visualization of a schedule s_p (left), and its end-shifted variant $\vec{s}_p$ with $\varphi_p = 2$ (right). Below each variant is the plot of the respective cumulative capacity $C_p(n)$. Figure derived from [30].

Next, we denote the basic flexcurve, which reflects the flow requirements for path P and frame size c, as

$$b_P(c) = \min_{p \in P} \sum_{\tau=0}^{h-c} 1_{\{C_p(\tau+c)-C_p(\tau)=c\}} \tag{4.2}$$

The flexcurve provides the number of possible arrangements for a flow with frame size c at the bottleneck schedule (cf. Lemma 4.1). The frame size c corresponds to the number of reserved slots for this flow's frame in the port schedule. The number of slots directly gives the time required to transmit the complete frame, as considered by the schedule. Therefore, this value depends on the schedule's time granularity and link speed. For instance, a nanosecond granularity reflects a time requirement of 1 ns per slot. A higher link speed results in fewer slots needed per frame bit. We assume a shared common link speed and granularity along path P. Note, the usage of minimum along the flow's path to identify the bottleneck schedule.

With $c \in \{1, \dots, h\}$ the flexcurve denotes all frame sizes along path P for cycles of h and undefined deadline without needing to recalculate the schedules along the path. Note, that flow-level schedules are considered, meaning transformed queue-level schedules and their potential queue isolation constraints are not reflected.

Lemma 4.1. *Given the basic flexcurve calculation*

$$b_P(c) = \min_{p \in P} \sum_{\tau=0}^{h-c} 1_{\{C_p(\tau+c)-C_p(\tau)=c\}} \tag{4.2}$$

and cumulative capacity $C_p(n)$, with the flow's frame interval set to the hyperperiod h, and deadline requirements disregarded , the expression $\sum_{\tau=0}^{h-c} 1_{\{C_p(\tau+c)-C_p(\tau)=c\}}$ accurately counts the number of valid arrangements for inserting a frame of size c into the schedule s_p.

Proof. Given a schedule s_p, a frame of size c is validly inserted into s_p if, from the starting position, the new frame reservation contiguously occupies slots in the schedule without overlapping with existing reservations, thus satisfying the flow's frame size constraint.

The expression $C_p(\tau + c) - C_p(\tau) = c$ checks whether the increase in cumulative capacity over the interval τ to $\tau+c$ precisely matches the size of the frame, c. This indicates that a contiguous block of time slots (τ to $\tau + c$) is available to accommodate the frame without any interruptions or overlaps with existing reservations. By summing over all possible starting times $\tau \in \{0, \dots, h-c\}$, the sum $\sum_\tau 1_{\{C_p(\tau+c)-C_p(\tau)=c\}}$ counts the total number of valid arrangements for frame size c at s_p.

Note that the last interval to be checked is from $h - c$ to h. By using $\vec{s}_p$ for the calculation of the cumulative capacity, we ensure that the cyclic nature of the schedule is adequately reflected (cf. Lemma 4.2). □

Lemma 4.2. *Given the hyperperiod h and $\mathcal{T}_{p,\beta}$ that gives the time point of the β-th free slot at the port schedule s_p. When calculating the cumulative capacity of port schedule s_p*

$$C_p(n) = \sum_\beta 1_{\{n \geqslant \mathcal{T}_{p,\beta}\}} \tag{4.1}$$

with end-shifted departure times $\vec{s}_p = (s_p^1 + \varphi_p, s_p^2 + \varphi_p, \dots, s_p^{X_p} + \varphi_p)$, $C_p(n)$ equals the total number of contiguous free slots without interruption at the cycle boundary.

Proof. The cumulative capacity $C_p(n)$ counts the number of free slots (capacity) up to slot index n. $C_p(n)$ only counts β up to the last free slot of s_p. At the most β reaches the hyperperiod h. It therefore does not account for repeated contiguous free slots that are interrupted by the cycle boundary.

Considering that the schedule repeats every h slots, the end-shifted sequence $\vec{s}_p$ is derived by adding the closing gap duration φ_p to the departure times, effectively shifting the observation window within the cyclic nature of the schedule without changing its periodicity. Therefore, the cumulative capacity (4.1), when calculated with the end-shifted sequence $\vec{s}_p$, equals the total number of contiguous free slots without interruption at the cycle boundary, ensuring that no contiguous free slots are interrupted by the cycle boundary. □

The formulation of (4.2) does consider overlapping arrangements. Restricting the formulation to not consider overlaps would reduce the encoded information that the basic formulation provides:

- **Number of Arrangements:** By definition, the flexcurve $b_P(c)$ provides the number of possible arrangements for a flow of size c along path P.
- **Residual Capacity:** The flexcurve encodes the residual capacity at the bottleneck schedule. $b_P(1)$ retrieves the number of possible arrangements for a frame of

size 1 along path P. All available slots are eligible because 1-slot is the smallest possible frame size that a schedule can support. Hence, the bottleneck's remaining capacity is returned.

- **Maximum frame size:** By Lemma 4.1, a positive value of $b_P(c)$ indicates that there is at least one possible arrangement along path P that can accommodate a new flow with frame size c. The value $\arg\min_c b_P(c)|_{b_P(c)>0}$ identifies the frame size c for which $b_P(c)$ is still positive. The use of arg min finds the frame size c that minimizes $b_P(c)$ under the constraint that $b_P(c)$ is positive, effectively identifying the largest frame size that is still eligible for inclusion, i. e., has at least one possible arrangement along path P. Therefore, the maximum frame size, still eligible for embedding, can be retrieved by

$$c_{max,P} = \arg\min_c b_P(c)|_{b_P(c)>0} \tag{4.3}$$

- **Disaggregations:** The flexcurve can be disaggregated to improve checking for admissibility and aggregated to foster incremental updates. Refer to Sections 4.1.1 and 4.1.2.

Numerical Example

To facilitate the better understanding of the basic flexcurve, we provide the following example. The example is derived from [30].

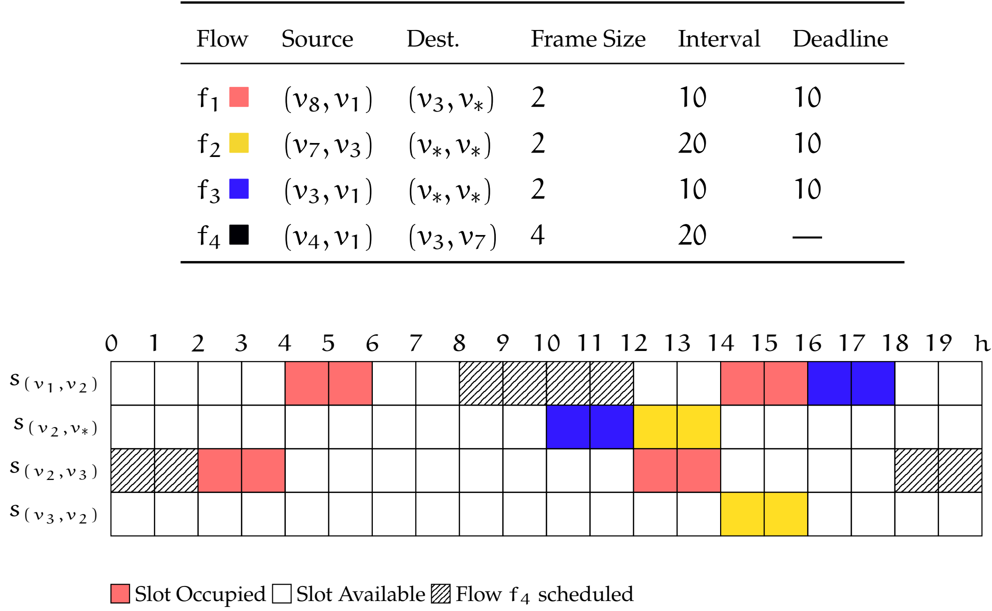
We define flows as illustrated in Figure 4.3. Each flow is characterized by a source and destination port identifier, along with specified frame size, interval, and deadline requirements. It is assumed that f_4■ has not yet been admitted to the network and lacks a specified deadline. A CNC has already configured the network and computed the schedules shown in Figure 4.3. Note that the provided schedule table is a partial view. The network's hyperperiod is defined as $h = 20$. The notation v_* signifies a node beyond the scope depicted.

We marked the future reserved slots of f_4■ with crosshatching. A routing overview for each flow is provided in Figure 4.4. Notably, there is a potential bottleneck at ports (v_1, v_2) and (v_2, v_*).

With the current port schedules, the flexcurve can be calculated for any arbitrary path. This path does not need to match currently active flow paths or even anticipated future paths. Any path of interest can be chosen.

For practical purposes, paths expected to contain flows are of particular interest. To simplify the example, we present three paths, with $P_\square$ denoting the path of flow $f_\delta\square$ within the scope of the depicted topology.

It is noteworthy that flexcurves for disjoint paths might be identical if their bottlenecks are at the same port. In this example, this applies to f_1■ and f_3■, both of which have an identical flexcurve for all frame sizes, limited by the bottleneck at port (v_1, v_2) at the selected path. We depict the flexcurves of flows f_1■, f_2■ and f_3■ in Figure 4.5. Notice, how after admission of f_4■, $b_{P_■}(c)$ and $b_{P_■}(c)$ are negatively affected. The



Flow	Source	Dest.	Frame Size	Interval	Deadline
f_1 ■	(v_8, v_1)	(v_3, v_*)	2	10	10
f_2 ■	(v_7, v_3)	(v_*, v_*)	2	20	10
f_3 ■	(v_3, v_1)	(v_*, v_*)	2	10	10
f_4 ■	(v_4, v_1)	(v_3, v_7)	4	20	—

Figure 4.3: Example: The figure shows the schedule allocation of three flows: f_1 ■, f_2 ■, and f_3 ■ across four ports. Their requirements are met in the TSN network. Flow f_4 ■ is scheduled for inclusion.

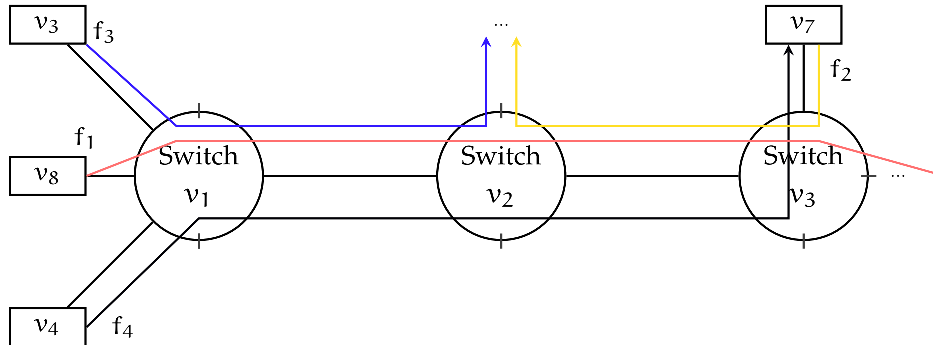


Figure 4.4: Example topology: It consists of three switches v_1 to v_3 and four talker nodes v_3, v_4, v_7 and v_8. The routed paths of flows f_1 to f_4 are depicted.

residual capacity $b_P(1)$ is affected by the frame size of f_4■. The maximum embeddable frame size $c_{max,P}$ is reduced as well. Whereas, $b_{P_■}(c)$ is completely unaffected. This is because f_2■ and f_4■ do not share a path, hence the bottleneck of f_2■ cannot shift.

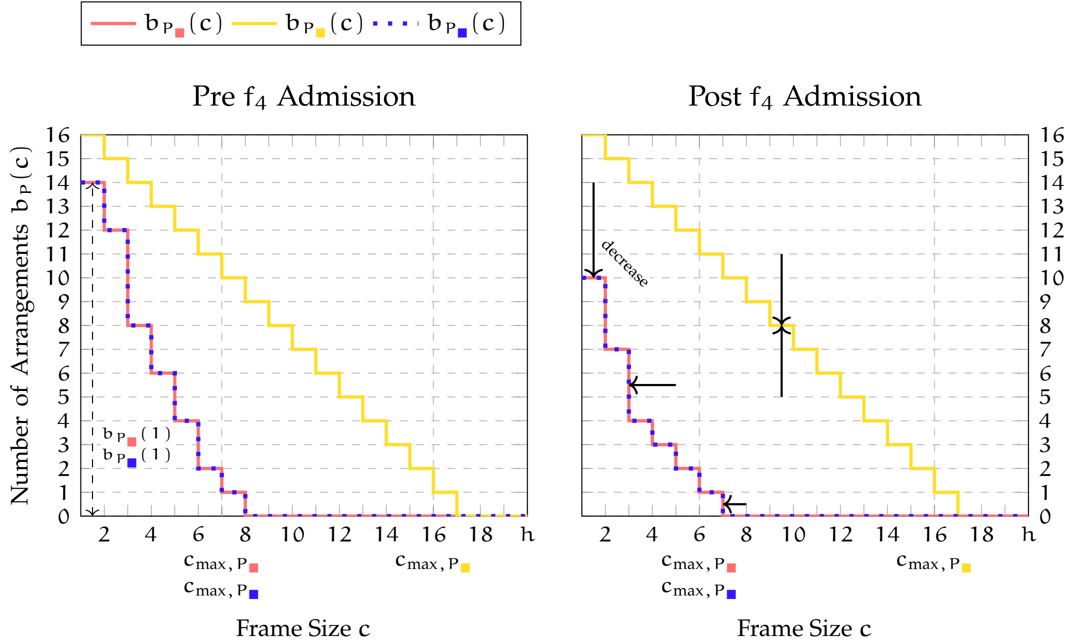


Figure 4.5: Example: Three flexcurves with given paths (cf. Figure 4.4), before (left) and after (right) admission of f_4■. Notice $b_{P_■}(c)$ and $b_{P_■}(c)$ share a bottleneck, and are affected after flow admission. Whereas, $b_{P_■}(c)$ is unaffected, while f_3■ and f_3■ are sharing ports. Figure derived from [30].

4.1.1 *Disaggregations for Admissibility Decisions*

The first and direct application of the flexcurve extending beyond merely quantifying flexibility are flow admissibility decisions. The flexcurve is defined to provide the number of possible arrangements for a specific flow. The parameters considered vary according to the specific version of the flexcurve utilized.

In subsequent discussions, we employ the basic definition of the flexcurve, denoted as $b_P(c)$. Nevertheless, it is possible to substitute this with the deadline-aware version, $b_P^d(c)$, which is detailed later in Section 4.1.3, or to explore alternative versions as discussed in Section 6.3, incorporating additional parameters as necessary.

Basic Flow Admissibility

To decide the admissibility of a flow with frame size c for path P, we apply the definition $b_P(c) > 0$. An admission is possible with any value greater than zero, subject to remaining flow requirements not mapped by the flexcurve's version.

A note on path selection: A flow f_δ may have multiple eligible paths that it can be routed through to the destination: $\mathfrak{P}_\delta = \{P_\delta^1, P_\delta^2, ...\}$. Each path can be checked for its eligibility: $\forall P \in \mathfrak{P}_\delta : b_P(c) > 0$. For a basic narrowing of the number of viable path

candidates, the path with the highest overall flexibility value – i. e., the largest area sum under the flexcurve – can be selected:

$$P_{\max,\delta} = \underset{P \in \mathfrak{P}_\delta}{\arg\max} \sum_{c} b_P(c) \tag{4.4}$$

It is important to note that predicting the change in flexibility value after the flow's actual admission is challenging, as the change in $b_P(c)$ depends on the flow's starting position assigned by the scheduler. Given the inability to directly determine the changes in $b_P(c)$, a refined path selection method should incorporate an approximation that admits the flow to path P. This method evaluates the effects on flexibility within the scheduling process without formal flow admission. Refer to Chapter 5 for a potential approach.

Additionally, selecting a path based on the current flexibility does not guarantee optimal utilization, since subsequently admitted flows may introduce cross traffic, thus rendering previous decisions suboptimal.

Multi Flow Admissibility

The following approach is restricted to the basic flexcurve $b_P(c)$.

A hard problem for heuristic schedulers is the admission of multiple flows in one atomic step. To the best of our knowledge, all heuristic TSN scheduling approaches for scheduled traffic, only support incremental single flow admissions. However, the ordering of flow admissions affects admissibility. *Badly* assigned starting times can block further admissions, which would have been possible if the next flow requirements were considered.

The basic flexcurve $b_P(c)$ can decide admissibility for multiple flows in the following way. Let $\mathbf{F} = \mathbf{f}_1, \dots, \mathbf{f}_u \subseteq F$ be the set of flows, each with their corresponding frame sizes $\mathbf{c}_1, \dots, \mathbf{c}_u$ and paths $\mathbf{P}_1, \dots, \mathbf{P}_u$, that are currently requested and dependent. Multiple flows are considered dependent if they share at least one common transmission port along their paths. Let $\mathcal{P} = \bigcap \mathbf{P}_\alpha$ represent the set intersection of all requested source/destination port pairs. This implies that if $\mathcal{P} \neq \emptyset$, then there exist dependent flows. Independent flows may be processed in parallel. Note that we assume that every port within the intersection $\mathcal{P}$ contains ports belonging to all $\mathbf{f}_\alpha \in \mathbf{F}$. If disjunct intersections exist, they need to be considered separately. Dependent flows can be checked for simultaneous admission by checking the concatenated frame sizes for admissibility at each shared port:

$$b_{\mathcal{P}}\left(\sum_{\alpha=1}^{u} \mathbf{c}_\alpha\right) > 0 \tag{4.5}$$

Note that $\mathcal{P}$ need not be a valid path. We use the flexcurve's ability to check ports independently. If (4.5) confirms possible admission at each shared port, we further need to check each flow's unique admissibility, as the individual path was not yet regarded:

$$\forall \alpha \in \{1, \dots, u\} : b_{\mathbf{P}_\alpha}(\mathbf{c}_\alpha) > 0 \tag{4.6}$$

If Eqs. (4.5) and (4.6) hold, the resulting admissibility condition suffices under the basic flexcurve model. This is because the assessment at shared ports accounts for the combined frame sizes, necessitating a contiguous slot availability. However, additional capacity may be found if inter frame fragmentation is possible.

We can check for possible fragmentations using a set of disaggregated flexcurves. The λ-th disaggregated flexcurve $b'_{\mathcal{P},\lambda}(c)$ can be obtained by recursively subtracting the canonical flexcurve $\tilde{b}_{\Delta}(c)$, essentially reducing the capacity for each flow $\mathbf{f}_\alpha$ to be included:

$$b'_{\mathcal{P},0}(c) = b_{\mathcal{P}}(c) \tag{4.7}$$

$$b'_{\mathcal{P},\lambda}(c) = b'_{\mathcal{P},\lambda-1}(c) - \tilde{b}_{\mathbf{c}_{\max,\lambda-1,\mathcal{P}}}(c) \tag{4.8}$$

The canonical flexcurve is given by

$$\tilde{b}_{\Delta}(c) = \max\{0, \Delta - c + 1\} \tag{4.9}$$

reflecting the staircase flexcurve, as if the schedule is empty with a period of Δ. We define $\mathbf{c}_{\max,\lambda,\mathcal{P}}$ analogous to $c_{\max,P}$ (4.3) as

$$\mathbf{c}_{\max,\lambda,\mathcal{P}} = \arg\min_{c} b'_{\mathcal{P},\lambda}(c)|_{b'_{\mathcal{P},\lambda}(c)>0}. \tag{4.10}$$

With $\mathbf{c}_{\max,\lambda,\mathcal{P}}$, the maximum frame size still eligible for the λ-th disaggregated flexcurve is retrieved. Therefore, $\mathbf{c}_{\max,0,\mathcal{P}} = c_{\max,\mathcal{P}}$. Each disaggregation step yields a canonical flexcurve of contiguous available slots, reflecting a single port flexcurve with an empty schedule and a period of Δ. Note that a canonical flexcurve cannot be mapped to a specific slot sequence; it represents continuous capacity. See also Section 4.1.2 for a discussion on the aggregation of canonical flexcurves.

Note the usage of $\mathcal{P}$ which denotes a *virtual* path. Essentially, the disaggregated flexcurve considers all ports that are shared. The virtual path matches the flow paths if and only if the complete path is shared. The subtraction is recursively applied u times, or until $b'_{\mathcal{P},\lambda}(1) = 0$ which indicates the residual capacity has been exhausted.

An example of a two step disaggregation for the flexcurve is depicted in Figure 4.6. The initial flexcurve (left, solid) is reduced by the 4th canonical flexcurve $\tilde{b}_{\mathbf{c}_{\max,0,\mathcal{P}}}(c) = \tilde{b}_4(c)$. In the next step (middle), the 3rd canonical flexcurve $\tilde{b}_{\mathbf{c}_{\max,1,\mathcal{P}}}(c) = \tilde{b}_3(c)$ reduces the residual capacity to zero (right). Given the presence of two canonical flexcurves, it can be concluded that the capacity at the bottleneck on the virtual path $\mathcal{P}$ allowed for a total of two independent contiguous available slots (gaps) to be utilized.

We can check for admissibility, by leveraging the canonical flexcurves, gained from the disaggregation process. As each canonical flexcurve represents contiguous available free slots, we can check each for admissibility, using the following heuristic. We assume a decreasing order of frame sizes, i. e., $\mathbf{c}_1 \geqslant \ldots \geqslant \mathbf{c}_u$ stands. We concatenate the frame sizes, from the largest to the smallest frame, until an admission cannot be achieved, i. e., there is insufficient capacity in the current disaggregation. The following Eq. (4.11) maps each disaggregation step λ to a set of indices $\mathcal{C}_\lambda$, that the canonical

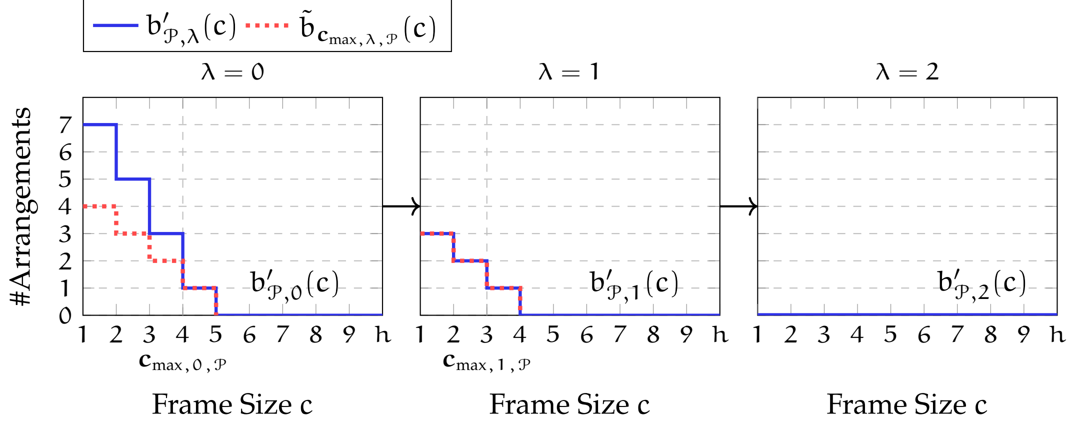


Figure 4.6: An example of flexcurve disaggregations is depicted. The initial flexcurve (left, solid) can be disaggregated twice until the residual capacity is exhausted. At each step, the canonical flexcurve (dotted) is subtracted, reducing the capacity by one maximum sized embedding. (For the large version of this Figure reference Figure A.4)

flexcurve $\tilde{b}_\Delta(c)$ can admit, excluding frame sizes of frames that have already been admitted at a prior step:

$$\mathcal{C}_0 = \emptyset$$
$$\mathcal{C}_\lambda = \left\{ \alpha \mid \tilde{b}_{\mathbf{c}_{\max,\lambda-1,\mathcal{P}}} \left(\sum_{i=\min\Gamma}^{\alpha} \mathbf{c}_i \right) > 0, \forall \alpha \in \Gamma := \{1, \dots, u\} \setminus \bigcup_{i=0}^{\lambda-1} \mathcal{C}_i \right\} \quad (4.11)$$

In the first step, $\mathcal{C}_1$, Eq. (4.11) checks up to which index α, the requested flows $\mathbf{f}_\alpha$ can be concatenated for admittance. Eligible indices are removed in the subsequent step $\lambda + 1$, and so on, until all flows can be admitted, or there are insufficient disaggregations. Note the similarities to Eq. (4.5), in checking the aggregation of flow sizes. Given that the canonical flexcurve encodes a contiguous number of free slots, an alternative approach involves verifying whether the aggregation $\sum \mathbf{c}_i \leqslant \mathbf{c}_{\max,\lambda-1,\mathcal{P}}$. This method simplifies implementation.

This approach employs frame size ordering as a heuristic for assigning frames to a canonical flexcurve. Optimally leveraging the total capacity of each canonical flexcurve would increase admission possibilities. However, this necessitates combinatorial methods, in which case a scheduler could initially be utilized. Other heuristics could also be applied, such as selecting the next frame based on the residual capacity remaining after placing a frame.

The approach to check for admissibility is detailed in Algorithm 1. We omitted early returns for readability. The algorithm first checks each flow for admissibility individually (lines 1 to 3). In the repeat-loop (line 6), the algorithm starts the disaggregation process, while simultaneously checking up to which index i the current disaggregated canonical flexcurve $\tilde{b}_{\mathbf{c}_{\max,\lambda,\mathcal{P}}}(c)$ can admit the flows $\mathbf{f}_i$ together (line 8). If an admission is no longer possible, the algorithm continues with the next disaggregation (line 12),

and repeats the aggregated admission check, now starting starting from index i, which indicates the last successfully admitted index.

If all flows are admitted, or if disaggregation is no longer possible, the algorithm exits the loop (line 13). The algorithm returns `True` if all flows are successfully admitted (line 15). If the algorithm cannot admit all flows, it returns `Null` to indicate an indeterminate result (line 17).

Algorithm 1 : Check the flow admissibility using flexcurve disaggregations. Derived from [30]

Input :

- Requested flows $\mathbf{f}_1, \dots, \mathbf{f}_u$, their corresponding paths $\mathbf{P}_1, \dots, \mathbf{P}_u$, sorted by their frame sizes $\mathbf{c}_1 \geqslant \dots \geqslant \mathbf{c}_u$. With $\mathcal{P} = \cap \mathbf{P}_\alpha$.

Result : True/False/Null indicating if the flows are admissible

1. **forall** $\alpha \in \{1, \dots, u\}$ **do** `// Individual admissibility (4.6)`
2.   **if** $b_{\mathbf{P}_\alpha}(\mathbf{c}_\alpha) \leqslant 0$ **then**
3.   **return** False
4. $\lambda \leftarrow 1$
5. $i \leftarrow 0$ `// Index of last flow possible to admit`
6. **repeat**
7. $c \leftarrow \mathbf{c}_{i+1}$ `// Concatenated frame size`
8.   **while** $\tilde{b}_{\mathbf{c}_{\max,\lambda,\mathcal{P}}}(c) > 0 \wedge i < u$ **do**
9. $i \leftarrow i + 1$
10. $c \leftarrow c + \mathbf{c}_{i+1}$
11.   `// Go to next disaggregation`
12. $\lambda \leftarrow \lambda + 1$
13. **until** $b'_{\mathcal{P},i}(1) = 0 \vee i > u$
14. **if** $i > u$ **then**
15.   **return** `True`
16. **else**
17.   **return** `Null`

4.1.2 *Aggregations for Quick Construction and Incremental Updates*

The flexcurve $b_P(c)$, as given in (4.2), is independent of the number of flows. However, calculating the flexcurve can be computationally intensive. To accumulate all values, considering the hyperperiod h, it is necessary to compute h values of the flexcurve. For each flexcurve value, the number of possible arrangements needs to be counted for each hop in the given path P with m hops. To count the number of possible arrangements, we can use $C_p(n)$ (4.1) within (4.2). This results in a very high computational runtime complexity, with hyperperiod h and path length m as a variable, of $O(mh^3)$ (cf. Lemma A.1). It is not unusual, when schedules offer nanosecond granu-

larity, to see hyperperiods of 1ms or more resulting in 10^6 slots or greater. Note, in implementations, optimized schedule data structures and dynamic programming approaches can reduce the runtime complexity of calculating the number of possible arrangements for a frame size c significantly. The path length is also bounded by the maximum hop distance between two nodes.

Computing the complete flexcurve up to h, might also not be necessary. When flows are limited to certain ranges, computing these ranges might suffice, greatly decreasing runtimes further. A reservation of the whole schedule's cycle is rather excessive. We also note, that values for the flexcurve can be computed asynchronously, meaning they can be precomputed offline, standing ready for potential flow requests.

However, when values are precomputed, any updates to the schedule invalidates all affected precomputations. In the following, we describe a possible computation for the flexcurve, that is optimized to enable rapid incremental updates. It also enables faster lookups of individual flexcurve values.

We employ an additional data structure, based on the schedule, to facilitate the retrieval of contiguous free slots (gaps) within the schedule. The sequence of gap starting times for schedule s_p is represented as $g_p = (g_p^1, g_p^2, \dots, g_p^{\Psi_p})$, along with their corresponding durations $\Delta_p = (\Delta_p^1, \Delta_p^2, \dots, \Delta_p^{\Psi_p})$. It is noted that the previously introduced φ_p, which indicates the duration of the gap following the last reservation, equals $\Delta_p^{\Psi_p}$ only when $g_p^{\Psi_p} + \Delta_p^{\Psi_p} = h$. The gaps in the mentioned sequences do not encode wraps, i.e., they conclude at the hyperperiod boundary h at the latest.

The flexcurve can be computed by aggregating the canonical flexcurves for each gap and limiting the value to the bottleneck port, effectively reversing the process of flexcurve disaggregation. An illustration of flexcurve aggregation is depicted in Figure 4.7. Each gap duration Δ_p^i within a schedule s_p defines its canonical flexcurve $\tilde{b}_\Delta(c)$. Per Lemma A.2 an empty schedule yields a canonical flexcurve. The gap-local canonical flexcurve specifies the number of feasible arrangements for a frame size c up to the duration of the gap. When $c = \Delta_p^i$, the gap is exactly filled, permitting only a single arrangement. The aggregation (4.14) of gap-local flexcurves results in a port-local flexcurve, indicating the aggregate number of arrangements for a frame size c at port p. Hence, this port-local flexcurve limits the perspective to a single port p, i. e., the path P comprises solely one port.

To reflect gaps correctly near the cycle boundary, we can base the gap sequence on the end-shifted schedule sequence $\vec{s}_p$, resulting in

$$\vec{g}_p = \begin{cases} (0, g_p^2 + \varphi_p, \dots, g_p^{\Psi_p - 1} + \varphi_p) & \text{for } \varphi_p > 0 \wedge g_p^1 > 0 \\ g_p & \text{otherwise} \end{cases} \tag{4.12}$$

$$\vec{\Delta}_p = \begin{cases} (\varphi_p + \Delta_p^1, \Delta_p^2, \dots, \Delta_p^{\Psi_p - 1}) & \text{for } \varphi_p > 0 \wedge g_p^1 > 0 \\ \Delta_p & \text{otherwise} \end{cases} \tag{4.13}$$

sequences. The approach is analogous to calculating the cumulative capacity $C_p(n)$ (4.1) using the end-shifted schedule $\vec{s}_p$. This is because the basic flexcurve values do not depend on the starting position of the cycle. The first and last gaps are merged

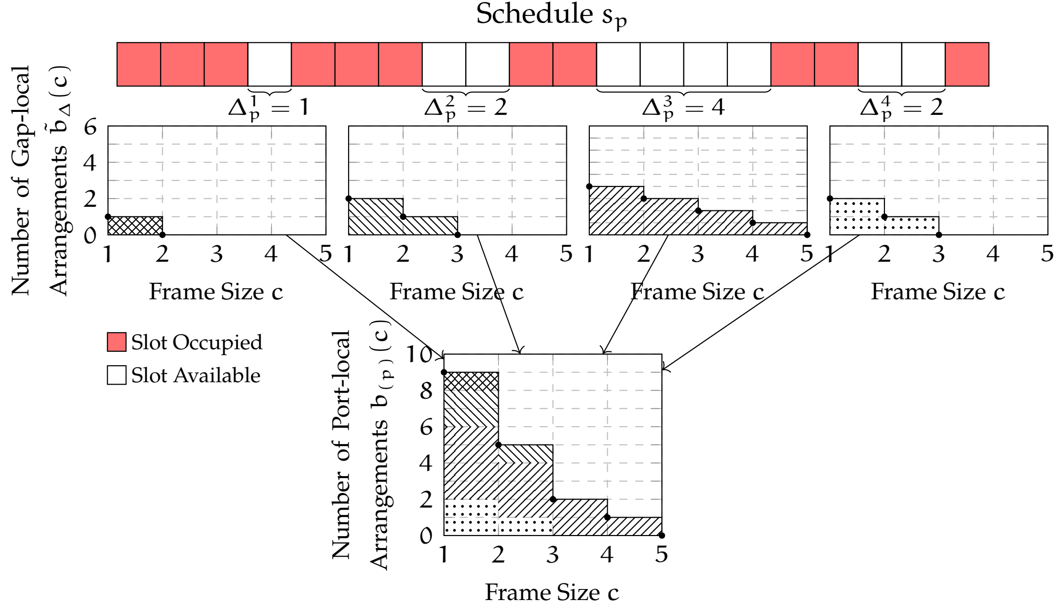


Figure 4.7: Example of a port-local aggregation of four canonical flexcurves, described by their gap durations $(\Delta_p^1, \Delta_p^2, \Delta_p^3, \Delta_p^4)$. The aggregated flexcurve is equal to a basic flexcurve reflecting only a single port $b_{\{p\}}(c)$. Figure derived from [26, 28]

only when there are free slots at both the beginning and end of the schedule, to reflect the contiguous number of free slots at the cycle boundary. See Appendix Figure A.1 for a visualization of the different end-shifted cases.

$$
\begin{aligned}
b_{\{p\}}(c) &= \sum_{i=1}^{\Psi_p} \tilde{b}_{\vec{\Delta}_p^i}(c) \\
&= \sum_{i=1}^{\Psi_p} \max\{0, \vec{\Delta}_p^i - c + 1\}
\end{aligned}
\tag{4.14}
$$

We note that the number of gaps in a schedule is usually much smaller than the number of slots $\Psi_p \ll h$. For paths with $m > 1$, per definition of the basic flexcurve (4.2), to reflect the bottleneck:

$$
b_P(c) = \min_{p \in P} b_{\{p\}}(c) = \min_{p \in P} \sum_{i=1}^{\Psi_p} \tilde{b}_{\vec{\Delta}_p^i}(c)
\tag{4.15}
$$

Thus, we obtain a basic flexcurve for path P by aggregating the gap-local flexcurves of each schedule along the path P.

The time complexity of creating the basic flexcurve for $c \in \{1, \ldots, h\}$, using gaps (g_p, Δ_p), with $\vec{\Psi}^P = \sum_{p \in P} \vec{\Psi}_p$ being the total number of gaps along the path P, is $O(hm\vec{\Psi}^P)$.

The shift from slot-level to gap-level generation of the flexcurve brings enormous complexity benefits. We also retain the ability to easily update the flexcurve[1]. When changes are introduced by the CNC, updates to the underlying flexcurve data structure become as simple as updating the schedule itself, provided that gap retrieval is supported. Otherwise, updating the gap sequence becomes necessary. However, only affected ports need to update the list. There are three possibilities after the scheduler assigns a single flow to a schedule s_p: (i) the gap is closed, resulting in the removal of the entry; (ii) the transmission time of the new flow either matches the starting time or connects to the end of the gap, in which case the gap duration is reduced by the frame size; or (iii) the flow is scheduled between the start and end of a gap, leading to the gap being split.

Bridging Aggregation and Disaggregation

It might seem evident that with aggregated canonical flexcurves we are capable of bypassing the disaggregation steps for admissibility decisions of multiple flows. However, it is important to note that the disaggregation step yields canonical flexcurves with respect to the overall path capacity. The capacity is bottlenecked by the schedule with the minimum residual capacity; however, frame sizes might still be limited by schedules before and after the residual capacity bottleneck. Gap-local flexcurves cannot be mapped 1-to-1 to disaggregated canonical flexcurves; therefore, not all gap-local flexcurves are eligible.

This is best demonstrated through an example: In Figure 4.8, we depict two schedules along a common path. Schedule s_{p_1} has a residual capacity of 6, whereas the highly fragmented schedule s_{p_2} has a greater residual capacity of 10 slots. The two resulting aggregated gap-local flexcurves are located below the schedule visualization. Even though s_{p_1} possesses a lower capacity, the gap duration $\Delta^2_{p_1} = 5$ is not eligible for use in a disaggregation process due to the frame size limitations of s_{p_2}. Disaggregating the final flexcurve (□) results in 6 canonical flexcurves with $\tilde{b}_1(c)$, for use in admissibility decisions. Notice how there is no mapping to specific gap-local flexcurves. Instead, the achievable frame sizes up to a maximum capacity are reflected.

4.1.3 *Deadline-awarenes*

Up until now, we have only considered the frame size c and h as flow parameters for admission to calculate flexibility values with the flexcurve and to check for admissibility. Evidently, the number of valid arrangements for admission can be reduced when additional constraints are considered. The *deadline* flow requirement is of special importance, as it is one of the key properties of the isochronous traffic type. We address this limitation by providing a formulation of the flexcurve that is deadline-aware.

1 Direct incremental updates are supported by the slot-level generation (4.2), however, depending on the implementation, schedule updates might trigger a complete recomputation, e.g., due to cache invalidations.

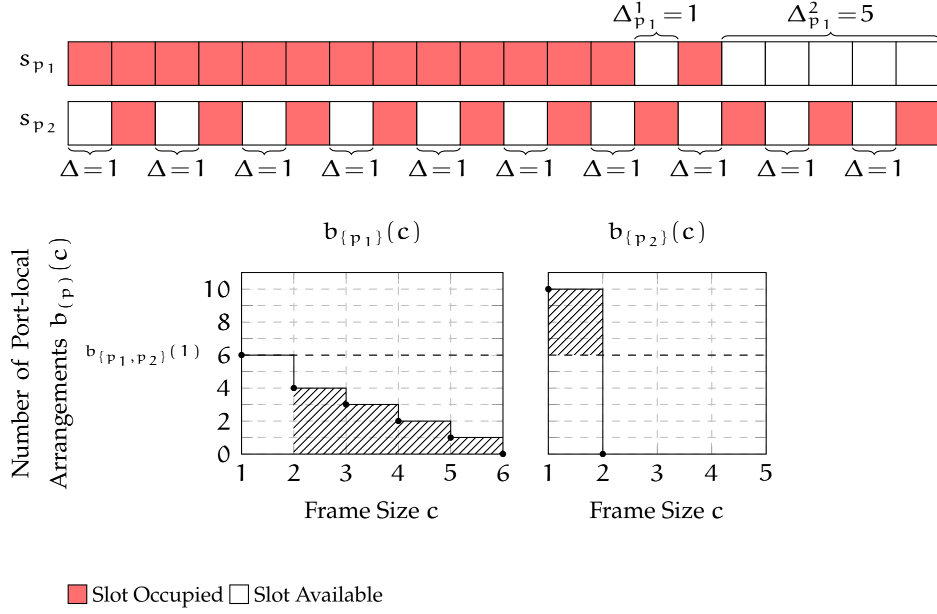


Figure 4.8: Example illustration of two schedules, s_{p_1} and s_{p_2}, along a common path with residual capacities of 6 and 10 slots, and their associated port-local flexcurves. Despite s_{p_1} having a lower capacity, the example highlights the ineligibility of the gap duration $\Delta^2_{p_1} = 5$ for the disaggregation process due to frame size constraints in s_{p_2}.

Because we are considering scheduled traffic, the delay a frame experiences is determined by the scheduler. It refers to the timespan from the first scheduled transmission time to the time the last transmission ends at the last scheduled port of the path. This delay must not exceed the deadline. Note that the transmission starts with the time-aware talker at the first scheduled transmission time at the earliest.

We extend the slotted schedule access notation with:

$$o_p(n) := \begin{cases} 1 & : \text{ slot is free} \\ 0 & : \text{ slot is occupied} \end{cases} \tag{4.16}$$

which denotes the occupancy of slot $n \in \{1, \dots, h\}$ in schedule s_p.

The delay between two time slots n, κ for any two consecutive ports $(p_\omega, p_{\omega+1})$ for a frame size of c is given by

$$t(n, \kappa) = \begin{cases} \kappa - n & : \kappa \geqslant n + c \\ h + \kappa - n & : \text{ otherwise} \end{cases} \tag{4.17}$$

The cycle of the schedule repeats every hyperperiod h, ensuring that in each cycle, there exists a transmission window for a slot. Consequently, the latest starting slot κ for a consecutive port is set to be one slot before the end of the previous transmission, denoted as $n + c - 1$.

A note on transmission delays: In real-world systems, additional delays are present. These delays stem from switch processing delays when frames arrive at a port, as well as propagation delays until bits arrive at the next hop. To enhance readability, we omit these additional delays and focus on the variable queueing delay. Propagation and link delays depend on the hardware and individual link properties and are typically considered constant, given the parameters. To account for the additional delays present at the affected hop, it would require to add the constant delay for the corresponding consecutive ports.

This results in queueing of the frame for nearly a complete cycle. Therefore the maximum possible delay between two consecutive ports results in:

$$t_{max} := t(n, n+c-1) = h + c - 1 \tag{4.18}$$

We describe a sequence of departure times for a flow with frame size c, along a path P with assignments $A = (a_1, \dots, a_m)$. It is important to note that this sequence does not necessarily constitute a valid schedule; it can describe potential candidates as well. Given a sequence of departure times A, we can aggregate the consecutive delays to calculate the end-to-end delay along path P with m hops.

$$T(A) = c + \sum_{i=1}^{m-1} t(a_i, a_{i+1}) \tag{4.19}$$

Note, we add the fame size c to account for the transmission duration at the last hop. The maximum end-to-end delay is given by $\max(T) = (m-1)t_{max} + c$.

Incorporating deadline requirements

A flexcurve that is deadline-aware, $b_P^d(c)$, should only consider possible frame arrangements on a given path P, if the frame size c, and *deadline* d, properties are satisfied. A flexcurve without deadlines calculates as many possible arrangements for flow admission as capacity allows, whereas a frame arrival deadline may restrict possible arrangements. Hence, tightening deadlines limits the flexibility of a configuration. Moreover, this is in line with our initial assumption: that the scheduled traffic configuration flexibility of a path is dependent on the specific flow properties.

Therefore, for a *deadline-aware flexcurve* of path P, with an end-to-end deadline d, the following applies:

$$b_P^d(c) \leqslant b_P(c) \tag{4.20}$$

$$b_P^d(c) = b_P(c) \text{ if } d \geqslant \max(T) \tag{4.21}$$

Any value of a deadline-aware flexcurve is smaller or equal to a basic flexcurve. For the proof refer to Lemma A.3.

In order to create a deadline-aware flexcurve, we first must be capable of determining the eligibility of the initial slot, and possible subsequent slots, based on whether they can meet the deadline requirements. This process bears a strong resemblance to

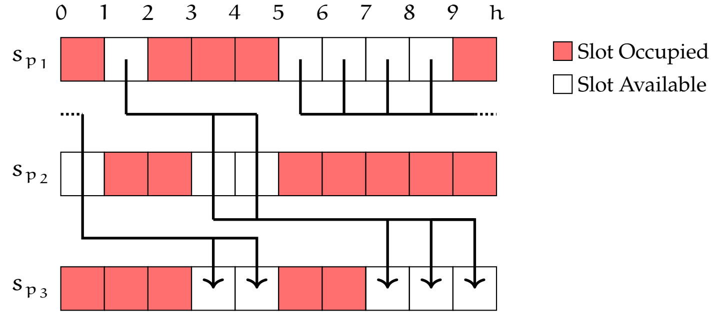


Figure 4.9: Example of starting, intermediary, and end assignments for a flow with frame size $c = 1$. Depending on the chosen start slot in schedule s_{p_1}, slots in consecutive port schedules up to the frame's end-to-end deadline are eligible for assignment.

a heuristic scheduler. In fact, this knowledge of eligibility can be utilized to schedule flows, due to the specific assignments required to verify the end-to-end delay. We elaborate on our approach to find slot eligibility in Chapter 5.

An illustration of such slot eligibility is depicted in Figure 4.9. Eligible slots must be selected based on the transmission start at the first schedule s_{p_1}. Up to the frame deadline, slots in subsequent schedules are either discarded or deemed eligible. Once a slot is eligible, it can be included in an eligibility set for frame size c, in the corresponding schedule's deadline-aware flexcurve computation.

The set of all eligible assignments for a flow with frame size c and deadline d is given by $A \in \overline{\mathcal{A}}$. We remind the reader that a flexcurve considers the bottleneck schedule. We reduce the set of assignments to a port p_ω with notation $a_\omega \in \overline{\mathcal{A}}^\omega$. The value of a deadline-aware flexcurve is reflected by the minimum number of eligible assignments along the path P:

$$b_P^d(c) = \min_{p_\omega \in P} |\overline{\mathcal{A}}^\omega| \tag{4.22}$$

for frame size c and deadline d, with $|\cdot|$ reflecting the cardinality of the set. Refer to Appendix Figure A.2 for a visualization of an example deadline-aware flexcurve.

4.1.4 *Holistic Flexibility View*

The flexcurve is a path-based metric; it does not measure overall network flexibility. To compare different configurations, the flexcurve inherently encourages selecting distinct paths for a path-based comparison. Nevertheless, a global view can sometimes be of interest when a comparison of complete networks is desired. The flexcurve formulation lends itself only very restrictively to describe this. We outline a possible approach to achieve a network-wide view.

If the set of viable routes in the network is known, then the flexcurve values for paths within this set of routes could be used in an aggregated fashion, i. e., given the set of all viable paths ρ, the sum

$$\sum_{P \in \rho} b_P(c) \tag{4.23}$$

would generate a network-wide flexcurve. To normalize the values and allow for comparison between networks with different numbers of routes, we can divide each value by the total number of viable routes $|\rho|$. However, we note the flexcurve is deliberately designed not to reflect overall network flexibility, as scheduled traffic is highly path-dependent. Distinct routes have no direct interference between each other.

One other network property that we intentionally disregard is cost and timing considerations for their impact on the flexibility level. By excluding cost considerations, we simplify the metric, which, due to the nature of scheduled traffic, already requires consideration of very specific requirements. It also allows us to focus on the main thesis goal of increasing the flexibility, by being able to measure the configurations inherent schedule flexibility.

Active Reordering to Improve Flexibility

One indirect application that is enabled by the flexcurve, in addition to measuring the level of flexibility, is the ability to precisely predict how a modification affects the schedule. In a static scenario and configuration, the modification of the active schedule, e. g., the shifting of transmission slots for certain flows, is neither intended nor necessary. There exists no notion of where existing slots should or need to be shifted to.

With the usage of the flexcurve, this changes, as now when a controller intends to modify the schedule, it is able to assess the modified flexibility impact, essentially improving the flexibility during operation. This might be utilized when flows leave the system. Of course, the actual shifting process of flows within a schedule is non-trivial and is out of scope of this thesis. Factors other than just flexibility need to be considered, such as the allowed jitter, or the impact on consecutive ports queue isolation.

Future Flow Assumptions

The flexcurve does not include assumptions of traffic pattern behaviors in the future. The level of flexibility might change depending on the traffic properties that are admitted in the future. Not including traffic assumptions has two main benefits:

- The flexcurve can support arbitrary traffic patterns; since there are no assumptions, it inherently supports various traffic properties. This is particularly important when deployments are unable to make predictions for future traffic inclusions.
- Reflecting future traffic patterns in the current flexcurve values would incorrectly skew the flexibility value if the future traffic patterns do not match. Hence, inclusion decisions might be made that are non-optimal, or even counterproductive.

We designed the flexcurve to avoid additional complexity in incorporating future traffic patterns and to focus on general assumptions. Nevertheless, incorporating future traffic assumptions has benefits. It provides the opportunity to better reflect the actual

flexibility level based on the traffic expected in the future. For example, when future traffic patterns are known, current configurations might offer zero flexibility, a state that is beneficial to accurately reflect.

Current Limitations

In this chapter, we have provided flexcurve formulations that reflect the level of flexibility based on the selected path and frame size. We have also included a method to incorporate deadline awareness. However, we did not extend the model to include deviating frame cycle times or an extension to consider TSN multi-mechanism support. For the latter, we have provided an extension in Section 6.3. The former is deferred to future work. With any additional extension, an increase in complexity is introduced. It highly depends on the individual requirements as to how accurately the flexcurve needs to reflect the level of flexibility.

4.2 EVALUATION

We evaluate the notion of flexibility and its applicability with different schedulers, as well as its runtime performance using the previously proposed formulations. Runtime evaluations were conducted using Python 3.12 on a MacBook Pro with an M1 Pro processor and 16GiB RAM

Within the first runtime evaluation (cf. Figure 4.10), we focus on the theoretical analysis from the previous section by comparing the computation runtime of initial flexcurve formulation (4.2) and aggregated formulation (4.15). The computation is constrained to a single schedule, which is populated in a stochastic manner, with slots being allocated based on a uniform distribution. Note, this approach produces high fragmentation, especially for low utilizations, and is not reflective of real-world schedules, where slots belonging to one flow are necessarily coherent. We visualize the effects on runtime based on the hyperperiod parameter. For each point, we sample $c \in \{500, \dots, 1000\}$ of the flexcurve.

The expectation is that the initial flexcurve formulation exhibits exponential runtime growth as the hyperperiod increases and the runtime reduces based on the utilization. We remind the reader that with an increase in slot utilization, the initial formulation, calculated at the slot level as presented in (4.2), leads to a decrease in the number of free slots; consequently, fewer slots are counted within the cumulative capacity formulation (4.1). For clarity, we omit the depiction of utilizations exceeding 100 slots in Figure 4.10.

However, the runtime is still heavily dependent on the hyperperiod, as evidenced by the basic flexcurve formulation's runtime complexity of $O(mh^3)$. With our evaluation assumptions, the basic formulation of (4.2) reduces to a complexity of $O(h^2)$, and that of the aggregated formulation (4.15) to $O(\Psi_{p_1})$. This behavior is depicted in Figure 4.10. The runtime analysis was performed ten times and averaged for the aggregated flexcurve and one time for the basic flexcurve. With a fixed number of slots, the runtime of (4.1) approaches a constant value. This is in line with our assumption that

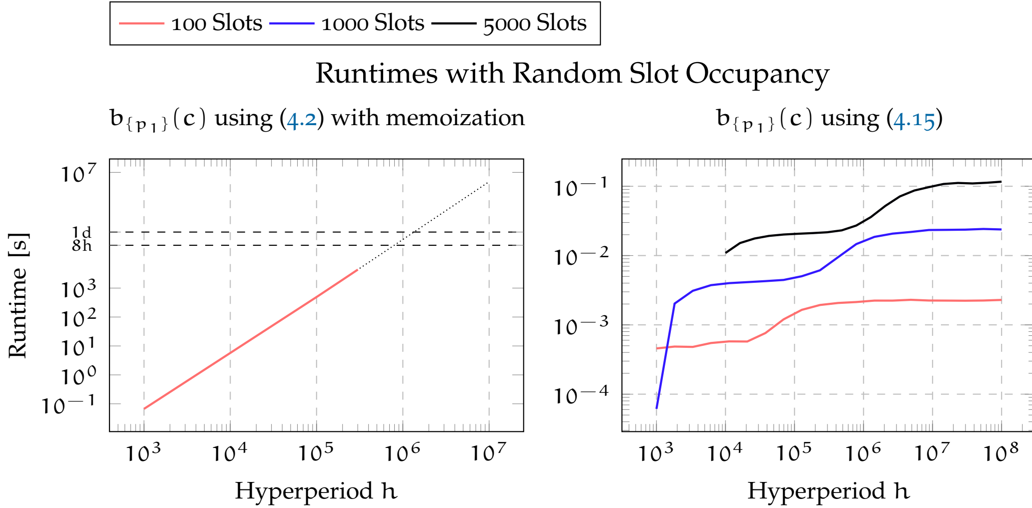


Figure 4.10: The experimental runtime analyses of implementations from initial flexcurve formulation (4.2), left, and aggregated formulation (4.15), right, are depicted. The runtime of (4.2) increases exponentially $O(h^2)$ with an increasing hyperperiod. The dotted line extension is the corresponding extrapolation. Conversely, the aggregated flexcurve (4.15) remains independent of the hyperperiod. As the hyperperiod increases, the initial random slot placement only minorly affects the gap count, explaining why the runtime approaches a constant time.

the number of gaps is the sole influencing factor. We used a memoization technique to cache the cumulative capacity for computation of (4.2).

The actual runtime behavior of how gaps affect the aggregated flexcurve (4.15) is investigated in Figure 4.11. The runtime demonstrates a linear correlation with increasing number of gaps. We conducted an evaluation of the runtime for up to 20,000 gaps within a schedule over a hyperperiod consisting of 50 million slots. The runtime analysis was conducted 25 times. The runtime is presented in the graph as the mean values, with error bars representing the standard deviation calculated using a delta degrees of freedom of 1. Such a schedule would permit at least 20,000 distinct transmission starts per cycle. The sampling of points remains at $c \in \{500, \dots, 1000\}$.

Next, we depict that the flexcurve can be used with output from various different schedulers for comparison purposes. Each scheduler necessarily outputs lists compatible with TSN gate control lists (GCLs); the lists describe the number of reserved slots and their reserved transmission starting times. This information can be used as schedule basis for the flexcurve calculation. This allows for a comparison of the output flexibility according to the flexcurve of different scheduling mechanisms or parameters. An example of such a comparison is given in Figure 4.12. Here, we compare three schedulers: an incremental flexcurve-based approach (discussed in the next chapter), an incremental satisfiability modulo theories (SMT)-based scheduler, a classical SMT-based scheduler, and a publicly available scheduler called TSNSched [83]. TSNSched is also based on SMT. We can observe that the incremental approaches result in the overall

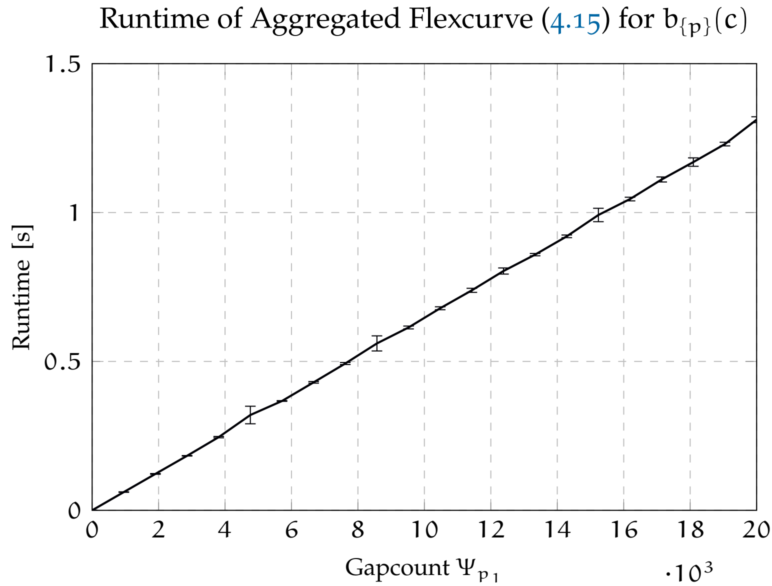


Figure 4.11: The experimental runtime analysis of (4.15). The runtime is linearly affected by the number of gaps within the schedule. The hyperperiod is fixed to 50 million slots.

highest flexibility values, directly followed by the classical SMT-based approach, and then by TSNSched. All schedulers were asked to schedule 10 flows with equal flow parameters for a given path. The schedules along this path are initialized empty. There are no direct optimizations regarding flexcurve values applied, to allow for direct comparisons. Due to the incremental nature of the scheduling, the incremental approaches are expected to achieve good performance. The SMT-based approach does not consider flexibility values. The scheduler decided to introduce fragmentation; hence, the schedule resulted in less flexibility. TSNSched also didn't introduce fragmentation; however, the schedule resulted in overall less residual capacity, due to padding.

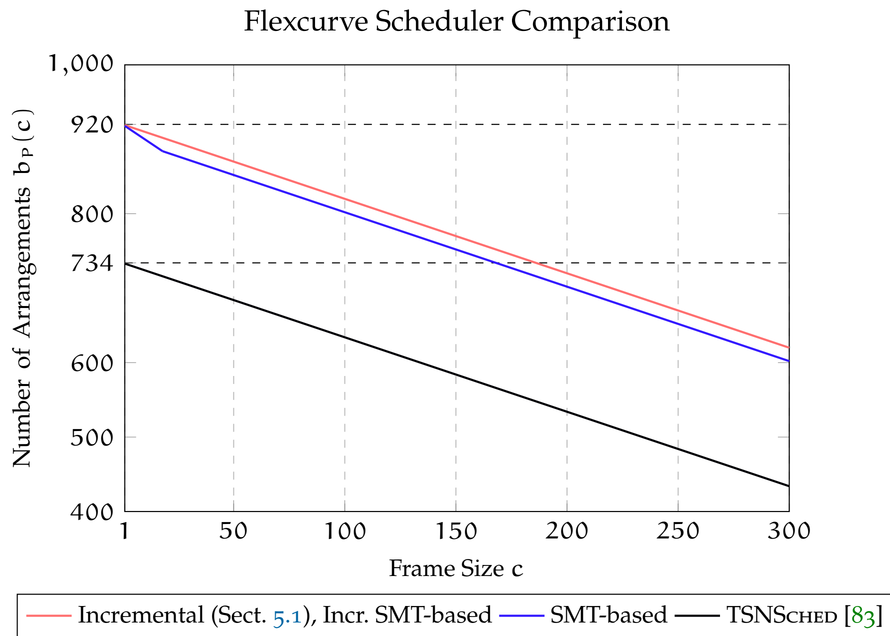


Figure 4.12: We compare the scheduling flexibility of three algorithms: a flexcurve-based, an SMT-based, and the open sourced TSNSCHED scheduler. It highlights that the flexcurve-based scheduler offers the highest flexibility according to the flexcurve metric, while the SMT-based scheduler shows reduced flexibility due to fragmentation, and TSNSCHED the least, due to introduced slot padding behavior.
Figure derived from [26, 28]

5 OPTIMIZATION

In this chapter, we apply the notion of flexibility, flexcurve, from the previous chapter to steer scheduling decisions for requested changes in Time-Sensitive Networking (TSN) configurations, thus addressing research goal 2. Additionally, we present a search algorithm designed to assist in creating a deadline-aware flexcurve. This algorithm is also applicable for the scheduling of individual flows.

5.1 FLEXCURVE-BASED SCHEDULING

In principle, any scheduler that can generate optimal flow-level schedules for TSN, including satisfiability modulo theories (SMT) and integer linear programming (ILP) approaches, could employ a flexcurve to enhance the scheduling process and improve the flexcurve values of the resulting schedules. However, integrating a flexcurve steering mechanism within heuristic schedulers might prove more practical. Classical constraint-based schedulers already consume substantial computing resources, and incorporating additional criteria increases both complexity and computational time.

In Section 4.1.3, we outlined a method for generating a deadline-aware flexcurve. A deadline-aware flexcurve necessitates the explicit knowledge of the scheduled time at each hop to accurately determine the resulting end-to-end delay. This knowledge is essential to construct a deadline-aware flexcurve. Subsequently, we describe how to identify eligibility candidates for a deadline-aware flexcurve and, in a second step, how to decide which candidate should be selected based on its flexcurve value. Finally, we apply a heuristic to incorporate the candidate selection directly into the candidate search algorithm itself.

5.1.1 *Eligibility Candidates for Deadline-aware Flexcurves*

Given a schedule s_p, a frame of size c can be validly inserted into the port schedule s_p if it occupies slots contiguously from the starting position and does not overlap with existing transmissions. This is reflected by the basic flexcurve (cf. Lemma 4.1). To additionally reflect frame deadlines, to create a deadline-aware flexcurve, the frame deadline d must also be considered. Therefore, a frame of size c can be validly inserted into s_p if the new frame transmission contiguously occupies slots in the schedule without overlapping with existing transmissions *and* the end-to-end delay does not exceed the deadline: $T(A) \leqslant d$. We note that this, similar to a basic flexcurve, allows for frames to be queued at each port.

We remind the reader that the set of all eligible schedule assignments for a flow with frame size c and deadline d along the path $P = (p_1, \dots, p_m)$ is given by $A \in \overline{\mathcal{A}}$. The schedule assignments $A = (a_1, \dots, a_m)$ used for the deadline-aware flexcurve must

satisfy the constraints imposed by the frame size and deadline. The deadline-aware flexcurve $b_P^d(c)$ depicts the number of arrangements at the bottleneck port, which correspond to the given frame size c and frame deadline d for the frame cycle period h. Thus, an algorithm designed to populate such an eligibility list must consider these constraints.

Initial Assignments

We first present an algorithm capable of finding a valid schedule assignment with the configuration's smallest possible deadline, allowing for queueing, for a single flow. The deadline is given by scheduled assignments with $T(A)$. Afterwards, we extend this algorithm to populate the eligibility list.

The management scenario described in Chapter 3 is assumed. Specifically, flows are requested according to their application requirements. The Centralized Network Configuration (CNC) selects a path, which is subsequently passed as a parameter to the algorithm that follows.

Initially, we apply a first-fit heuristic to schedule the flow. A first-fit heuristic operates by assigning the first viable slots in each port schedule to the flow's frame. The transmission time for each consecutive ports is assigned, at the earliest from the transmission end of the previous port's schedule. An example of such a first-fitting is depicted in the upper group of Figure 5.1. This enables a rapid identification of flow port assignments A that satisfy the basic flexcurve constraints. Thus, if the first-fit heuristic finds a flow assignment, only the frame size constraint is satisfied. If the first-fit heuristic is unable to schedule the flow, the given path has insufficient residual capacity to accommodate the new flow.

The assignment results in a specific end-to-end delay of $T(A)$, which might exceed the flow's deadline d. Note, we continue to omit switch processing and propagation delays from our considerations to simplify the notation. Compare with Figure A.3 for the inclusion of processing and propagation delays.

It is necessary to consider other possible transmission starting times in a_1 to determine if they can achieve quicker delays.

Shifting Assignments

We propose a search algorithm (Algorithm 2) designed to identify a valid assignment A that achieves the minimum possible end-to-end delay on the requested path $\mathbf{P}$. This algorithm ensures compliance with the requested application's deadline $\mathbf{d}$ and frame size $\mathbf{c}$, whenever feasible. It works by iteratively adjusting the starting transmission time a_1 to the right, shifting by one slot at each iteration (Line 4). Initially, assignments are determined using a first-fit heuristic, allowing for an early return if the delay constraint is already met by this preliminary assignment (Line 1). This incremental shift strategy is key, as it ensures that potential latency improvements are found that depend on the initial position of a_1.

To maintain the temporal order and immediacy of consecutive port assignments, the algorithm verifies after each shift of a_1 that subsequent assignments $(a_2, \dots, a_m)$

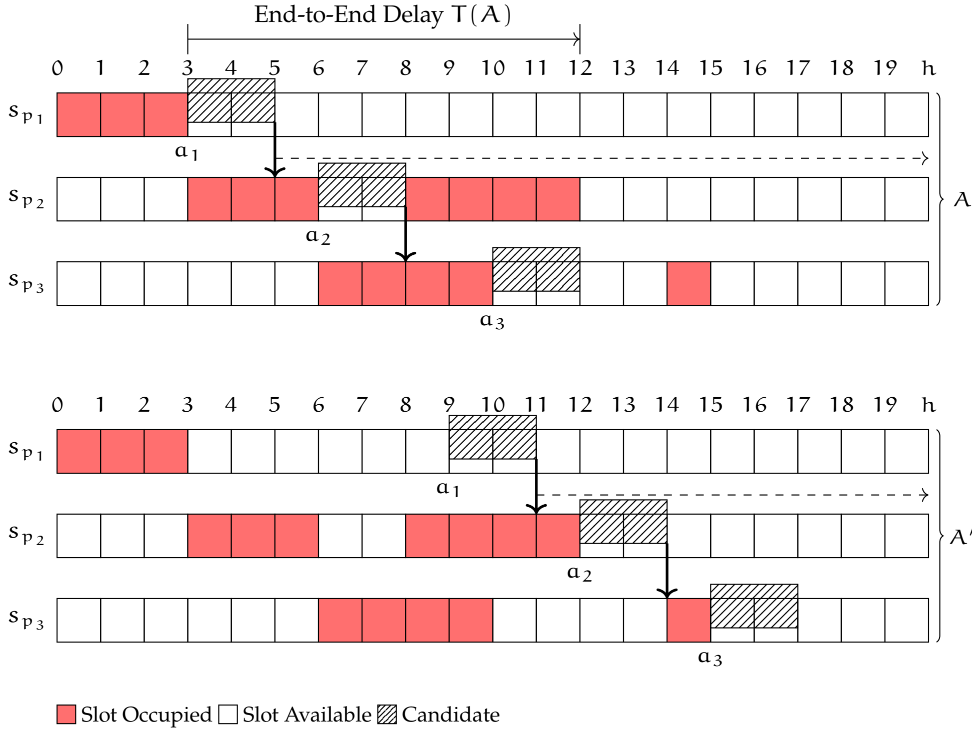


Figure 5.1: Two example states of Algorithm 2. The search algorithm shifts the starting position of the candidate a_1 within the schedule to the right. The following assignments (a_2, a_3) always follow their corresponding preceding assignment. s_{p_1} reflects the talker source port. Red slots are reserved by different flows.
Figure derived from [26, 28].

do not commence earlier than their predecessors (Line 10). Further, if the identified gap for an assignment becomes too narrow to accommodate the frame, the algorithm bypasses this invalid starting time in favor of the next feasible gap (Lines 6 and 13). The process returns successfully when it finds a path assignment A that meets the deadline criteria (Line 15). If no such assignment can be identified, the algorithm concludes that a viable solution is unattainable (Line 17). To consider the cyclic nature of each port schedule, we can repeat the reservations in each schedule beyond the hyperperiod. Resulting assignments need to be adjusted for the relative start afterwards (using the modulo operation $a_i \mod h$). An example of this shifting operation is depicted in Figure 5.1. The assignments are depicted as white crossed boxes. The initial assignment (A) is given by a first-fit heuristic scheduler. After several shift operations in s_{p_1} for a_1, the updated assignments (A') are reflected. Consecutive assignments $(a_2, \dots, a_m)$ are *pushed* by their corresponding previous assignment (black arrow down). When *virtually* extending the schedules, we can also simplify the end-to-end delay calculation. As there is no longer a need to compute each consecutive delay explicitly, we can immediately retrieve the end-to-end delay by

$$T(A) = a_m - a_1 + c \quad \text{for non wrapping extended schedules.} \tag{5.1}$$

Algorithm 2 never revisits slots, and in the worst case needs to shift a_1 up to hyperperiod h times. Each consecutive assignment $(a_2, \dots, a_m)$ only needs a fixed number of operations to follow the preceding assignment. Hence, the algorithm has a worst-case runtime complexity of $O(hm)$. For a hyperperiod h and path length m.

Algorithm 2 : Find flow position eligibility candidate. Derived from [26, 28].

Input :

- Requested flow $\mathbf{f}$, its corresponding path $\mathbf{P}$, deadline $\mathbf{d}$, and frame size $\mathbf{c}$.
- For each port along $\mathbf{P}$, initial assignments A of the requested flow $\mathbf{f}$. Assignments are denoted as $A = (a_1, \dots, a_m)$.
- For each port along $\mathbf{P}$, schedule gaps $g_{p_1}, \dots, g_{p_m}$ and gap durations $\Delta_{p_1}, \dots, \Delta_{p_m}$.
- Mapping from a port assignment to a specific gap: $\text{Gap}(a_\omega) = i$, where i is the largest index such that: $g^i_{p_\omega} \leqslant a_\omega$

Note: To reduce notational overhead when considering cycle boundaries and potential wraps, we extend the port schedules and their corresponding gaps beyond the hyperperiod. Additionally, gaps in the list are filtered to be able to contain the requested frame size $\mathbf{c}$*. We also omit some failure cases for readability.*

Result : A/False

```
1  if T(A) ⩽ d then       // Return the initial first-fit if the deadline is met.
2      return A
3  while a_1 < h do       // Shift initial assignments up to hyperperiod.
4      a_1 ← a_1 + 1
5      i ← Gap(a_1)
6      if g^i_{p_1} + Δ^i_{p_1} − a_1 < 0 then    // Check if current gap is large enough.
7          // Otherwise, use the next viable gap
8          a_1 ← g^{i+1}_{p_1}
9      for ω in 2, ..., m do
10         if a_ω < a_{ω−1} + c then              // Assignments follow previous.
11             a_ω ← a_{ω−1} + c
12             i ← Gap(a_ω)
13             if g^i_{p_ω} + Δ^i_{p_ω} − a_ω < 0 then   // Move to next gap if needed.
14                 a_ω ← g^{i+1}_{p_ω}
15     if T(A) ⩽ d then
16         return A
17 return False
```

Finding eligibility candidates

Algorithm 2, in its unmodified form, only returns one eligible candidate assignment. However, to create the deadline-aware flexcurve, we need all slot eligibility candidates. This allows to accurately reflect the eligibility number at the bottleneck needed for creation of the flexcurve. Two modifications are required to yield such a list.

1. Instead of early termination upon finding a solution, the discovered solution is appended to a result list. The algorithm then continues to iterate over a_1 up to the hyperperiod for each execution.

2. Secondary shifts must be introduced to adjust the consecutive assignments with $(a_2, \ldots, a_m)$ up to the allowed end-to-end delay $T(A)$ of the deadline d.

It is tempting to omit the second modification; however, doing so might underestimate the value of the deadline-aware flexcurve, as slots that might still be eligible under the deadline constraint are not considered. Consider the scenario depicted in Figure 5.2, where eligibility for the initial assignment is limited. In the initial description, the algorithm would already stop at this initial state, overlooking the eligible slots of the second schedule s_{p_2}. However, when eligibility candidates are shifted until the deadline is reached, every eligible slot is visited, and the resulting candidate list accurately reflects the assignments. The secondary shift modification, detailed in Algorithm 3, can also be seen as a reverse shifting pass. This occurs because the primary shift progresses along the designated path, while the secondary shift operates in the opposite direction. This reversal is essential to accommodate the shifting up to the frame deadline d, thereby guaranteeing that successive assignments do not commence prior to the completion of their predecessors. This modification can be introduced after the primary shift in Algorithm 2 at Lines 9-14. The assignments are reverse-shifted in Algorithm 3 up to the last eligible slot that is below or at the deadline (Line 10), with eligible assignments being added to the result set (Line 11). We denote the result set of eligible assignments found by Algorithms 2 and 3 as $\overline{\overline{\mathcal{A}}}$.

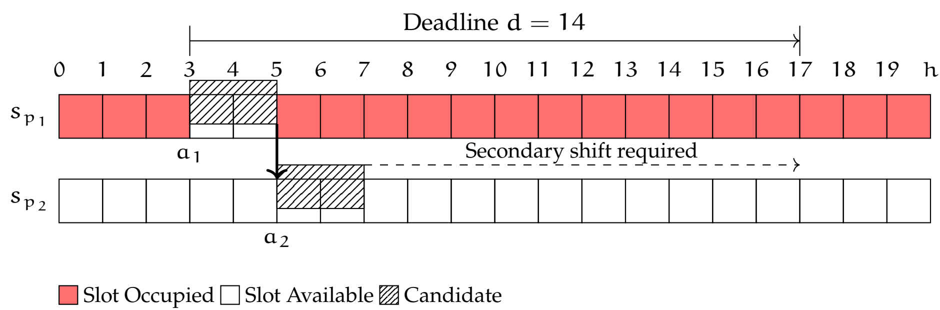


Figure 5.2: Example of the secondary shift modification for Algorithm 2. The discovery of additional eligible slots beyond the initial assignment, up to the allowed deadline, is enabled.

It is important to note that the modification, as outlined in Algorithm 3, does not identify every possible combination of eligible slots $\overline{\mathcal{A}}$. For the creation of the deadline-

aware flexcurve (cf. Section 4.1.3), it is sufficient that an eligible slot is included at least once.

Algorithm 3 : Secondary shift. Modification for Algorithm 2.

Input :

This algorithm modifies Algorithm 2 with a secondary shift to find all candidates up to the deadline. It is applied after the primary shift (Lines 9-14). Inputs are adopted from Algorithm 2.

Note: We also omit some failure cases for readability.

```
1  for ω in m, ..., 2 do                                   // Shift in reverse order.
2      shiftedUntilDeadline ← False
3      while ¬shiftedUntilDeadline do
4          i ← Gap(a_ω)
5          if g^i_{p_ω} + Δ^i_{p_ω} − a_ω < 0 then          // Move to next gap if needed.
6              a_ω ← g^{i+1}_{p_ω}
7          else
8              a_ω ← a_ω + 1
9              // Shift until the deadline or the begin of the following
           assignment.
10         if ω = m and T(A) ≤ d or ω < m and a_ω + c ≤ a_{ω+1} then
11             A̿ ← A̿ ∪ {A}
12         else
13             shiftedUntilDeadline ← True
```

5.1.2 *Eligibility Candidate Selection*

The process extends beyond merely generating a list of eligible assignments. It further requires the selection of a suitable candidate to optimize the resulting flexcurve value.

The remainder of this Subsection 5.1.2 is taken verbatim from Section 4.3.1 of [28]. The notation is adapted to align with the overall thesis notation.

In this subsection, we illustrate the update of the deadline-aware flexcurve as depicted in Figure 3.2. The depicted computation of the new schedule can be carried out using different methods with or without taking the flexcurve as a constraint. In the following, we show how to obtain nearly optimal schedules in terms of the schedule flexibility *after* embedding an incoming flow request. Indeed, different proper flow embeddings may well lead to different schedule flexibility values *after* the embedding. To this end, we show next how to choose the assignment candidate for a given flow request such

that the deadline-aware flexcurve is maximized. A further constraint is to achieve this in an fast and incremental fashion.

Since any $A \in \overline{\overline{\mathcal{A}}}$ is a valid assignment candidate according to the flow requirements $(\mathbf{P}, \mathbf{c}, \mathbf{d})$, any appropriate candidate could be directly admitted. However, since $\overline{\overline{\mathcal{A}}}$ only gives the current deadline aware flexcurve value $b_P^d(c)$ for the currently deployed schedule, the value of the resulting flexcurve after an admission is unknown. To find the updated value of the deadline aware flexcurve *after* candidate admission, the recreation of $\overline{\overline{\mathcal{A}}}$ based on the post-admission schedule is required.

The selection of an optimal candidate, i.e., A_k that maximizes $b_P^d(c)$ after admission requires repeating the deadline-aware flexcurve computation $|\overline{\overline{\mathcal{A}}}|$ times. This approach is only optimal within $\overline{\overline{\mathcal{A}}}$, as there may be a better assignment candidate, that is not included in the saved candidate set $\overline{\overline{\mathcal{A}}}$. Finding the optimal candidate is infeasible in production scenarios as even for simple scenarios the computation time quickly rises to multiple hours (cf. Figure 5.8 and Table 5.2) if the scheduling granularity or period requires a high amount of slots. Therefore, we propose an alternative approach by pruning the initial set $\overline{\overline{\mathcal{A}}}$ using the prior calculated arrangements of the candidates, to generate a near-optimal solution. Being able to remove candidates in $\overline{\overline{\mathcal{A}}}$ that would not occur after a candidate's admission directly gives the new flexcurve's value with $\min_{p \in P} |\overline{\overline{\mathcal{A}}}^p|$, without applying the modified embedding-search algorithm (Algorithms 2 and 3) again.

This pruning operation with a candidate $(q_1, \ldots, q_m) \in \overline{\overline{\mathcal{A}}}$ removes each $A_k \in \overline{\overline{\mathcal{A}}}$ where q overlaps with A_k. Note that it is sufficient for an element removal to detect an overlap at any port along the path. An overlap occurs when two compared assignment candidates occupy at least one identical time slot. More specifically: $(q_\omega) \in \overline{\overline{\mathcal{A}}}$ overlaps with another candidate A with a_ω along path $P = (p_1, \ldots, p_m)$, when the following is true:

$$\bigvee_{\omega \in \{1,\ldots,m\}} \left\{ \begin{matrix} a_\omega \leqslant q_\omega < a_\omega + \mathbf{c} \\ a_\omega + \mathbf{c} \geqslant q_\omega + \mathbf{c} > a_\omega \end{matrix} \right\} \tag{5.2}$$

For example, with set $\overline{\overline{\mathcal{A}}} = \{A_1, A_2, A_3\}$, and with A_1 overlapping A_2, the three pruning operations could result in the following three new sets:

$$\overline{\overline{\mathcal{A}}}_{A_1} = \{A_3\} \qquad \overline{\overline{\mathcal{A}}}_{A_2} = \{A_3\} \qquad \overline{\overline{\mathcal{A}}}_{A_3} = \{A_1, A_2\}$$

By pruning the result set, we remove all embeddings that could be selected instead of the candidate but are invalid for selection as the slots are now occupied. We therefore reduce the number of viable assignment candidates. Now, we assign the updated deadline-aware flexcurve values to each pruned set. For example, given

$$\min_{p_\omega \in P} \left|\overline{\overline{\mathcal{A}}}^\omega_{A_1}\right| = 4 \qquad \min_{p_\omega \in P} \left|\overline{\overline{\mathcal{A}}}^\omega_{A_2}\right| = 4 \qquad \min_{p_\omega \in P} \left|\overline{\overline{\mathcal{A}}}^\omega_{A_3}\right| = 12$$

we select A_3 as the near-optimal assignment candidate.

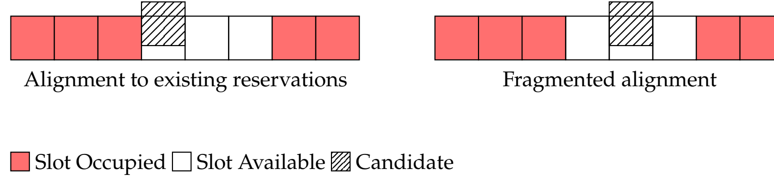


Figure 5.3: It is possible to prune fragmented candidates, as their resulting flexcurve value is always lower than that of aligned assignment candidates.
Figure derived from [26, 28].

This incremental approach requires $O(|\overline{\overline{\mathcal{A}}}|^2)$ operations as we are not further isolating good candidates, but it is much faster than finding the optimal solution. Flexcurve values are higher if less gaps are produced at equal utilization. Less gaps result in a higher count of possible arrangements, which the flexcurve value reflects. We can therefore pre-prune viable candidates by only pruning the result set $\overline{\overline{\mathcal{A}}}$ with candidates which align to or after existing schedule entries at any port-schedule. This pre-pruning is allowed because aligned assignment candidates will always result in less fragmented schedules thus higher flexcurve values (cf. Figure 5.3). While the worst-case complexity remains at $O(|\overline{\overline{\mathcal{A}}}|^2)$ our empirical results in Table 5.2 show an improve in runtime performance, as fewer candidates are checked in total. The significance of the runtime benefit depends on how many assignment candidates can be pre-pruned.

5.1.3 *In-place Scoring*

We can avoid the pruning process outlined in the previous section 5.1.2, if we are able to modify Algorithm 2 by incorporating a scoring heuristic. This scoring heuristic is intended for the selection of the best candidate while $\overline{\overline{\mathcal{A}}}$ is being populated.

We propose the following scoring mechanism for a candidate A, along path P and with a frame size c. The mapping from a port assignment to a specific gap is given by the function $\mathrm{Gap}(a_\omega) = i, \quad$ where i is the largest index such that: $g^i_{p_\omega} \leqslant a_\omega$.

$$\mu(\omega) = \min \begin{cases} a_\omega - g_{p_\omega}^{\mathrm{Gap}(a_\omega)} \\ g_{p_\omega}^{\mathrm{Gap}(a_\omega)} + \Delta_{p_\omega}^{\mathrm{Gap}(a_\omega)} - a_\omega + c \end{cases} \tag{5.3}$$

$$\max_{p_\omega \in P} \mu(\omega) \tag{5.4}$$

$$\sum_{p_\omega \in P} \mu(\omega) \tag{5.5}$$

For each candidate A, we apply Eq. (5.4) and keep track of the current best (i. e., the lowest value). The equation yields the maximum value of $\mu(\omega)$ along the given path P. Eq. (5.3), $\mu(\omega)$, calculates the distances to the beginning and end of the current gap of a_ω and returns the shortest distance to either end. This scoring mechanism essentially aims to minimize the distance of the candidate to either gap border. If a new candidate results in a lower maximum value according to Eq. (5.4), we additionally

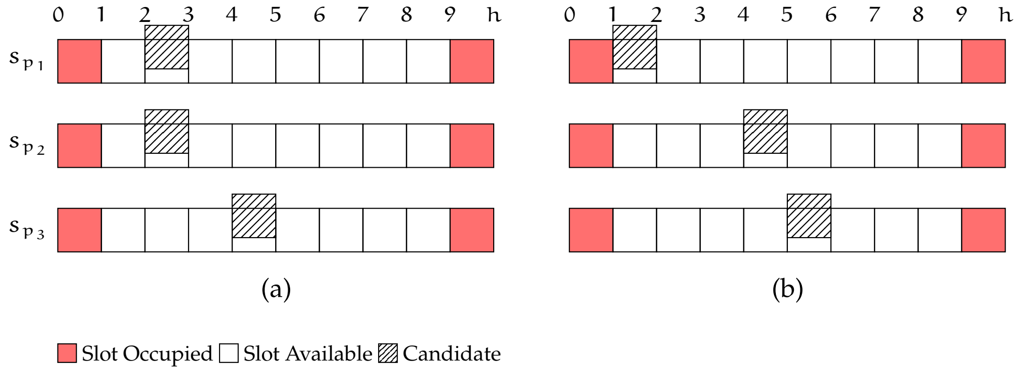


Figure 5.4: The example illustrates an in-place scoring outlier. The candidate placement in (a) has the same maximum distance to either end of the gaps with $\max \mu(\omega) = 3$ as (b). However, the total distance in (b) is higher, with $\sum \mu(\omega) = 6$, than in (a), where $\sum \mu(\omega) = 5$, due to the presence of more severe outliers.

apply Eq. (5.5). If the sum total, i. e., score, of the distances is lower, the current best candidate is updated. By employing this two-step approach, we ensure not to select bad outliers that have a low distance score but exhibit overall worse placements. We illustrate this behavior in Figure 5.4. The left (a) candidate placement has the same worst-case distance of three slots as the right (b) placement. However, the total distance in (b) is higher than (a), which would result in (a) being selected over (b).

Incorporating in-place scoring into the candidate selection can be seamlessly integrated into the $\overline{\overline{\mathcal{A}}}$ population process of Algorithm 2 itself, thereby eliminating costly post-processing times.

5.2 PATH SELECTION

Only a few approaches in TSN scheduling incorporate path selection as an inherent part of their scheduling process (cf. Section 2.3). Similarly, the scheduling strategy outlined with Algorithm 2, as discussed in Section 5.1.1, does not consider multiple eligible paths either.

We remind the reader that, as described in Section 4.1.1, the basic flexcurve can be utilized for making path selection decisions based on the current value of the flexcurve. This principle can analogously be applied to a deadline-aware flexcurve $b_P^d(c)$ with path P and frame deadline d. However, we noted the challenge in predicting the flexibility changes for the selected path.

When computing a deadline-aware flexcurve, given by the eligibility list $\overline{\overline{\mathcal{A}}}$,

$$b_P^d(c) = \min_{p_\omega \in P} |\overline{\overline{\mathcal{A}}}^\omega| \tag{4.22}$$

we can utilize the eligibility candidate selection process, as mentioned in Section 5.1.2, to approximate the new value of the flexcurve. The path for a flow f_δ with multiple eligible paths $\mathfrak{P}_\delta = \{P_\delta^1, P_\delta^2, ...\}$, can then be selected based on approximated expected

changes. This is done using the pruned eligibility list with candidate assignment $A \in \overline{\overline{\mathcal{A}}}$ for a path P, with frame size c and deadline d, using

$$P_{\max,\delta} = \arg\max_{P \in \mathfrak{P}_\delta} \sum_{c} \min_{p_\omega \in P} |\overline{\overline{\mathcal{A}}}_A^{\omega}| \tag{5.6}$$

Essentially selecting the *best* path with admission of the candidate.

5.3 QUEUE ASSIGNMENTS

As part of the hardware abstraction layer, that is part of the controller described in Section 3.2, we need a way to deploy the flow-level port schedules to a supported TSN mechanism. A mechanism able to support this is IEEE Std. 802.1Qbv [42], *Enhancements for scheduled traffic*, also known as Time Aware Shaper (TAS).

The remainder of this Section 5.3 is taken verbatim from Section 4.2 of [28], with some adaptions for clarity. The notation is adapted to align with the overall thesis notation.

While Algorithm 2 can find a valid embedding within the given port schedules, the resulting schedule may be infeasible for an immediate deployment with a TAS-capable device. The reason for this lies in the fact that TAS works by following the gate control list (GCL) of each output port to open and close priority queues for the egress. If packets are scheduled with a no-wait-constraint one queue is sufficient for scheduling as arriving packets are immediately processed and forwarded to the next hop. In our case, Algorithm 2 finds suitable embeddings which may require the packet to queue for a finite and predetermined amount of time before the scheduled time arrives, while still respecting any deadline requirements of the flow. This waiting capability requires flows to be isolated from each other, either by arrival time or by sorting into different queues. Otherwise flows can lead to false or premature forwarding.

Lack of isolation might occur when new flows arrive to the same queue before present flows can egress. Therefore, specific queue allocations are required, in addition to the scheduled time points A.

So far, we considered the requirement to utilize s_p at the slot level, i.e., to check whether slots are occupied or not. This was sufficient to find viable schedule embeddings and flexcurve values. To assign queue identifiers to A for deployment in a TAS switch (cf. Figure 2.2), we need to also consider the *separable flows* within s_p and their corresponding arrival and queue allocations at this port. There are χ_p departure flow instances within the schedule s_p. A flow instance s_p^i has a queue(s_p^i) identifier, an arrival(s_p^i) time relative to the start of s_p and duration(s_p^i) of occupancy at port p.

To assign viable queues for A, the returned embedding result of Algorithm 2, we need to make sure the chosen queue for a port along the path is (i) part of the available scheduled traffic queue set (cf. Figure 2.2) and (ii) not currently occupied by other flows already waiting at the port.[1]

1 To the best of our knowledge, current TSN switches do not allow changing the Priority Code Point (PCP) code, i.e., the priority code of a TSN flow, hence all chosen queue identifiers must be identical.

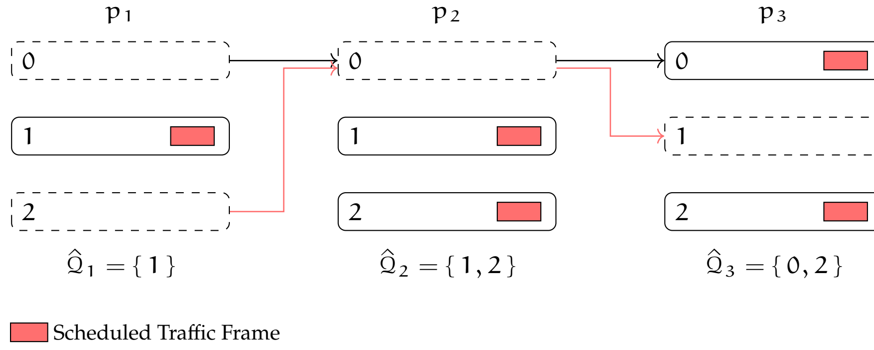


Figure 5.5: Example: At the time a_ω, the scheduled traffic queues at port p_ω are displayed, along with their occupancy. Only the empty queues (dashed) at the port are available for enqueuing to guarantee flow isolation. The example shows that a single common queue identifier cannot be used among all ports, but it is possible to find an assignment of queue identifiers for that path. Note that the corresponding implementation in state-of-the-art switch hardware requires the ability to rewrite the PCP field.
Figure derived from [28].

We assume a shared common queue set $\mathcal{Q}$ for scheduled traffic among all ports. Along the path $(p_1, \ldots, p_m)$, the occupied queue set for a port p_ω is given by $\hat{\mathcal{Q}}_\omega$ at time $a_{\omega-1}$ for the duration of the flow c as the chosen queue must be available at the start time of transmission at the previous port $p_{\omega-1}$. The first port p_1 is disregarded as it is part of the sender and we assume there are no incoming packets forwarded for switching purposes. The intersection of the occupied queue set with the common queue set gives the possible queue set $\mathcal{Q}'_\omega = \mathcal{Q} \cap \hat{\mathcal{Q}}_\omega$ of free queues at the desired port p_ω. This is depicted in Figure 5.5. Note that state-of-the-art TSN switches are not able to change the flow PCP code along a path. Hence, we find in this case that for a given flow the queue identifier is identical along the path. This queue identifier is chosen out of the set of possible queues that is given by the intersection of all possible queues $\bigcap_{\omega=1}^{m} \mathcal{Q}'_\omega$. This restriction is also depicted in Figure 5.5 as the black arrow.

To create the occupied queue set $\hat{\mathcal{Q}}_\omega$ at port p_ω all identifiers of occupied queue are combined into a set:

$$\hat{\mathcal{Q}}_\omega = \bigcup_{i \in [\chi_p]} \text{queue}(s^i_{p_\omega}) \quad \text{if } a_{\omega-1} \text{ intersects } s^i_{p_\omega} \tag{5.7}$$

An assignment $a_{\omega-1}$ intersects the flow instance s^i_p at port p_ω if the following two intervals intersect:

$$[a_{\omega-1}, a_{\omega-1} + c] \bigcap \left[\text{arrival}(s^i_{p_\omega}), \text{arrival}(s^i_{p_\omega}) + \text{duration}(s^i_{p_\omega})\right]$$

In other words, this condition arises if for any duration of the residing flow instance s^i_p the new flow to be embedded is scheduled to wait at port p_ω simultaneously. For example, if s^i_p represents a schedule entry that arrives at 10 and lasts at the port for 3 slots, then the interval $a_{\omega-1}$ intersects s^i_p if $a_{\omega-1} = 9$ and $c = 4$.

5.4 EVALUATION

New metrics or algorithms introduce different computational loads or additional computational overheads. Hence, it is important to consider the timing requirements and algorithmic complexity of the proposed concepts. The computational impacts on steering the scheduling process and path selection may not be tolerable for the network's users. Runtime evaluations were conducted using Python 3.12 on a MacBook Pro with an M1 Pro processor and 16GiB RAM. The timing requirements need to be experimentally evaluated on different topologies and with different parameters, such as network utilization, flow deadlines, frame sizes, and intervals. The evaluated topologies should reflect real-world scenarios as well as fabricated worst-case examples.

Flow and Topology Assumptions

For our runtime evaluations, we use two different topologies: First, a line-topology with four hops, reflecting a worst-case scenario, as each added flow interferes with the others, meaning each flow is added to the bottleneck. Second, a machine topology (cf. Figure 5.6), reflecting a real-world scenario [28]. The complex machine topology consists of a three-switch hierarchical layout, with three switches at the center, each with three subsections in a line topology. Each subsection switch has six end-nodes attached; one end-node is the designated machine programmable logic controller (PLC). Each node is time-aware and either a TSN-switch or an end-node/PLC. The number of flows within the machine-network results in a total of 106 flows. All end-nodes transmit flows to the designated PLC, and the PLC transmits flows to all other end-nodes as well. In Table 5.2, we list the flow properties that are used in the following evaluations.

Table 5.1: Flows used for evaluation. Derived from [26, 28].

Topology	Period [ms]	Deadline [ms]	Size [bytes]	Avg. Pkt Size [bytes]
Line	1.0	1.0	100	100
Machine	1.0	0.5	60 to 300	180

Scheduling

We first evaluate the runtime of Algorithm 2 for incrementally scheduling flows, as outlined in Section 5.1.1, i. e., for incoming single-flow requests, given the flow requirements in Table 5.1. We compare the runtime across the two given topologies with a classical SMT-based approach and an incremental SMT-based approach. The classical SMT approach schedules all flows simultaneously and is not incremental by design. This approach cannot guarantee online adaptations due to potentially shifting transmission times. In contrast, the incremental approaches fix the previously scheduled flow transmission times, thus only needing to find transmission times for the

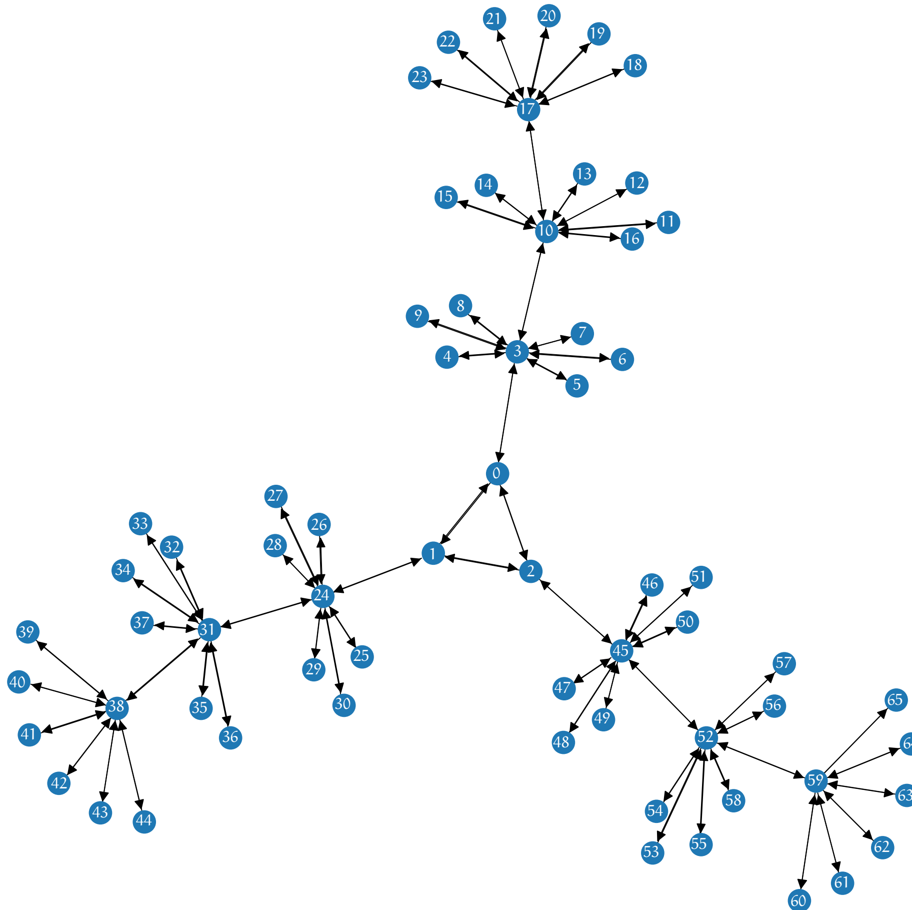


Figure 5.6: Depicted is the machine topology used for evaluation purposes. Nodes are numbered. The topology consists of three sections, three subsections and six end-nodes. One end-node is designated to act as the network's controller. Figure derived from [26, 28].

new flows. The empiric runtime results depicted in Figure 5.7 indicate that the classical approach, for a small number of flows, is still fast compared to the incremental approaches, with sub seconds runtimes. However, the classical SMT-based approach quickly escalates to a significant amount of computational time, reaching hourly runtimes with only 50 flows. This is the expected behavior of this holistic constraint-based approach. Both incremental approaches demonstrate linear runtime behavior up to the observed number of flows. The classical SMT-based approach was run one time for each depicted flow count, whereas the incremental approaches were averaged over 10 runs per added flow.

Eligibility Candidate Selection

Continuing, we evaluate the runtime of the eligibility candidate selection, as outlined in Section 5.1.2. As the flexcurve is path-based, we depict the scenarios in Table 5.2 for

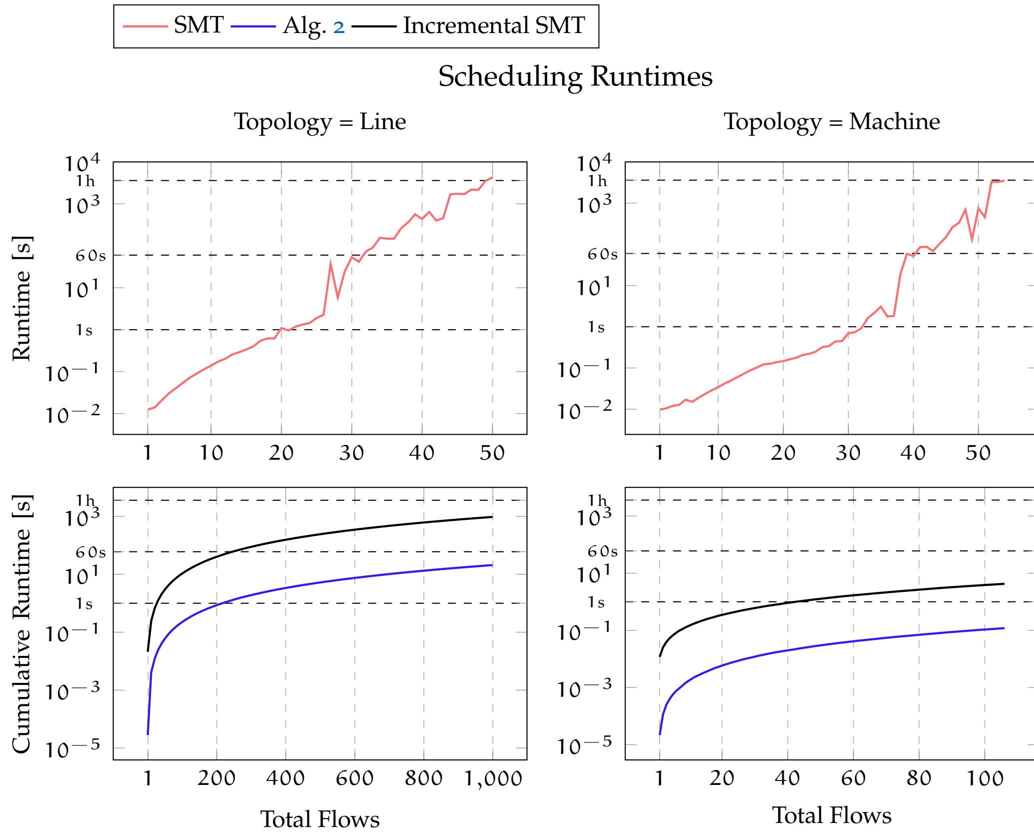


Figure 5.7: Scheduling Runtime comparisons of both line and machine topology. The classical SMT scheduling approach quickly rises to hourly runtimes, due to scheduling all flows together. The incremental SMT and incremental Algorithm 2 approach provide significant runtime advantages. Figure derived from [26, 28].

the line topology with 1000 slots for each schedule. We evaluate three scenarios: Scenario (i) *First Fit / 10 Flows*, admits 10 flows along the path with no fragmentation due to the first-fit admission strategy. Scenario (ii) *Random / 50 Flows* has 50 flows admitted with random frame transmission times, resulting in 35% utilization. Scenario (iii) *Random / 100 Flows* is analogous to (ii) with the same utilization of 35%, albeit with 100 flows. Increased fragmentation and utilization result in fewer candidates, improving the selection performance due to fewer candidates in the eligibility list. The optimal recomputation approach of the deadline-aware flexcurve is not affected by the number of eligibility candidates, as it is applied independently again and works with the scheduling state itself.

Deadline-aware Flexcurve

With the eligibility list (cf. Section 5.1.1), a deadline-aware flexcurve can be described. In Figure 5.8, we depict the empirically gathered runtimes for the computation of one

Table 5.2: Computation times for eligibility candidate selection, compared to a complete recomputation. Table derived from [26, 28].

Scenario	$\|\overline{\overline{\mathcal{A}}}\|$	Recompute [s]	Selection (5.1.2) [s]	w/ Prepruning (5.1.2) [s]
First Fit / 10 Flows	8115	126.79	24.02	7.41
Random / 50 Flows	3186	191.88	8.54	7.48
Random / 100 Flows	2376	448.00	4.75	4.64

point on the deadline-aware flexcurve. We consider four different utilizations: 10%, 25%, 35%, and 50%. The schedules along the evaluated path (2–6 hops) are allocated in a uniformly random manner, leading to a very high fragmentation. As the eligibility list needs to consider each starting position, the runtime increases as more slots are associated with a schedule. This behavior is observed to be linear and matches the expectation from Algorithm 2 and its secondary shift extension given in Algorithm 3. The number of hops in the path influences the slope of the runtime graph. This also matches the expectation, because the algorithm needs to shift more schedule candidates along a longer path. Also, note that with increased utilization, the runtime decreases. This is because fewer slots are eligible; therefore, fewer slots need to be checked for the creation of the eligibility list in Algorithms 2 and 3.

In-place Scoring Comparison

Utilizing our prototype controller implementation, as documented in Appendix A.1 and part of [27], we enabled the in-place scoring selection mechanism (cf. Section 5.1.3) for application in a scheduling scenario. This strategy is compared with random flow assignments and the first-fit strategy. The in-place scoring heuristic prefers candidates that align with existing schedule slot entries. The in-place scoring heuristic exhibits behavior akin to the first-fit strategy when schedule gaps are progressively filled by new frame reservations. In contrast, the random strategy assigns frames to empty slots in a uniformly random manner, ignoring deadline requirements and serving as a baseline.

We define the following scenario: the visualization is restricted to a sampled cumulative flexcurve using frame sizes $c \in \{100, 110, 120, 130, \ldots, 240\}$ along the selected path (p_1, p_2, p_3). The underlying network topology is depicted in Figure 5.9. Admitted frames may traverse two potential paths: Path A, which is the selected and visualized path, and Path B, which introduces cross traffic at the second port p_2. Along with the sampled cumulative flexcurve, we also depict a visualization of the number of slot fragments. The selected path is initialized with 40 randomly placed flows with frame properties relfected from the scenario.

The sampled frame sizes reflect a subset of the frame sizes from flows that are requested for admission. The frame sizes admitted are uniformly and randomly distributed between 100 and 250 slots. The frame periods are set to 15,000 slots with an end-to-end deadline of the same duration.

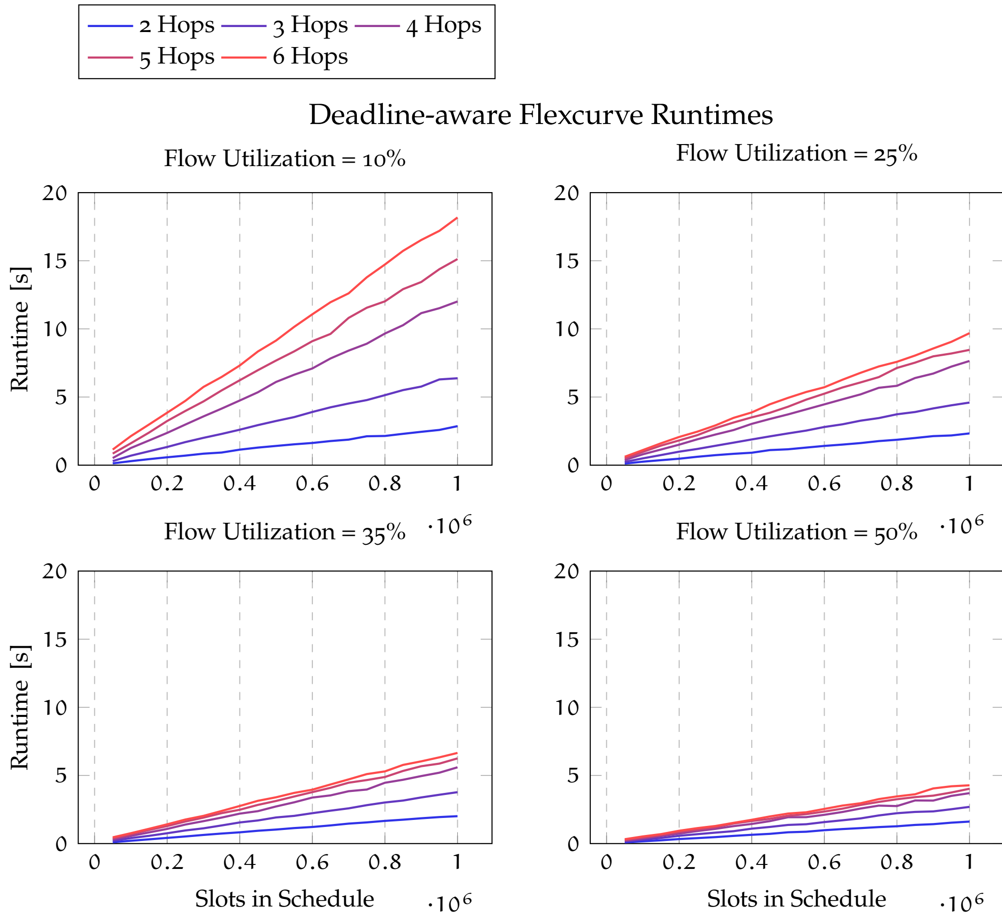


Figure 5.8: Depicted is the deadline-aware flexcurve runtime. Schedules are randomly allocated with varying utilizations; as utilization increases, the runtime decreases due to a reduced number of slots available that must be considered for the eligibility list. Figure derived from [26, 28].

We conduct a simulative evaluation of the behavior of three strategies across two dynamic scenarios. In Scenario 1, flow admission requests are clustered before a cluster of flow evictions occurs. When a flow is evicted, a random active flow is removed. Scenario 2 features fewer large clusters of admissions and evictions, leading to more continuous dynamism. The specific sequence of flow requests for Scenarios 1 and 2 is detailed in Appendix A.4.

The first scenario, depicted in Figure 5.10, illustrates a decrease in flexibility upon accepting flow admission requests, and an increase when flows are evicted. The second scenario, visualized in Figure 5.11, shows that the fragmentation remains on similar levels under a random strategy throughout the scenario, due to shorter admission clusters. In contrast, flexibility is distinctly impacted due to the gradual increases in admitted cross-traffic, which creates a bottleneck at the second port. It is noteworthy that the first-fit strategy effectively tracks the changes of the in-place scoring selection.

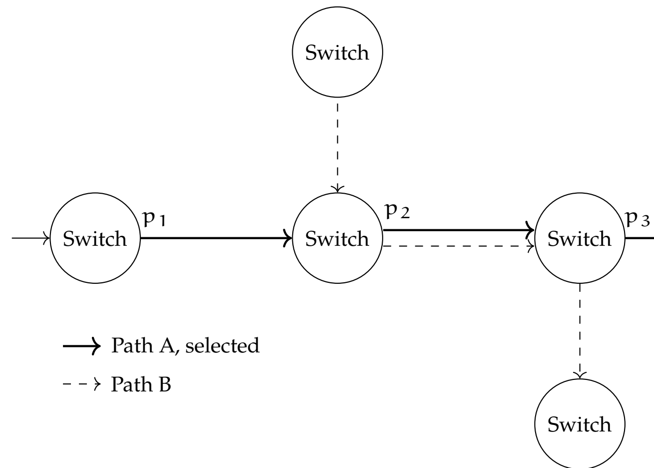


Figure 5.9: Topology of the controller scenario. Figure derived from [27].

Regarding the performance of the greedy first-fit algorithm relative to in-place scoring, several factors contribute to its effectiveness. Firstly, the in-place score approximation targets a single point of the flexcurve using deadline and frame size, whereas the visualization utilizes a cumulative flexcurve derived from multiple frame sizes. Secondly, the uniformity of frame periods ensures favorable flexcurve outcomes when fragmentation is minimal. This is achieved by the first-fit strategy's ability to locate the first viable spot, thus minimizing gaps. Lastly, the in-place selection focuses on the requested paths, rather than the selected visualization of path (p_1, p_2, p_3).

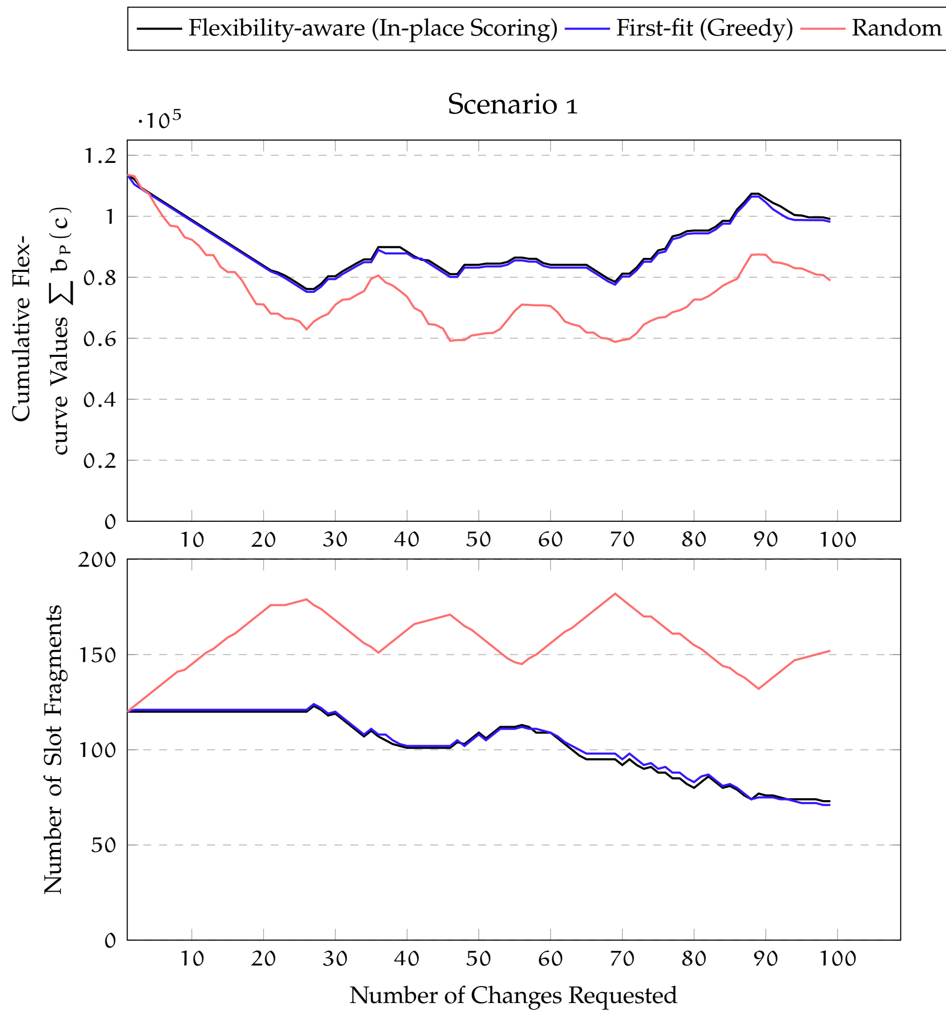


Figure 5.10: Flow requests with scenario 1 (Appendix A.4). Figure derived from [27].

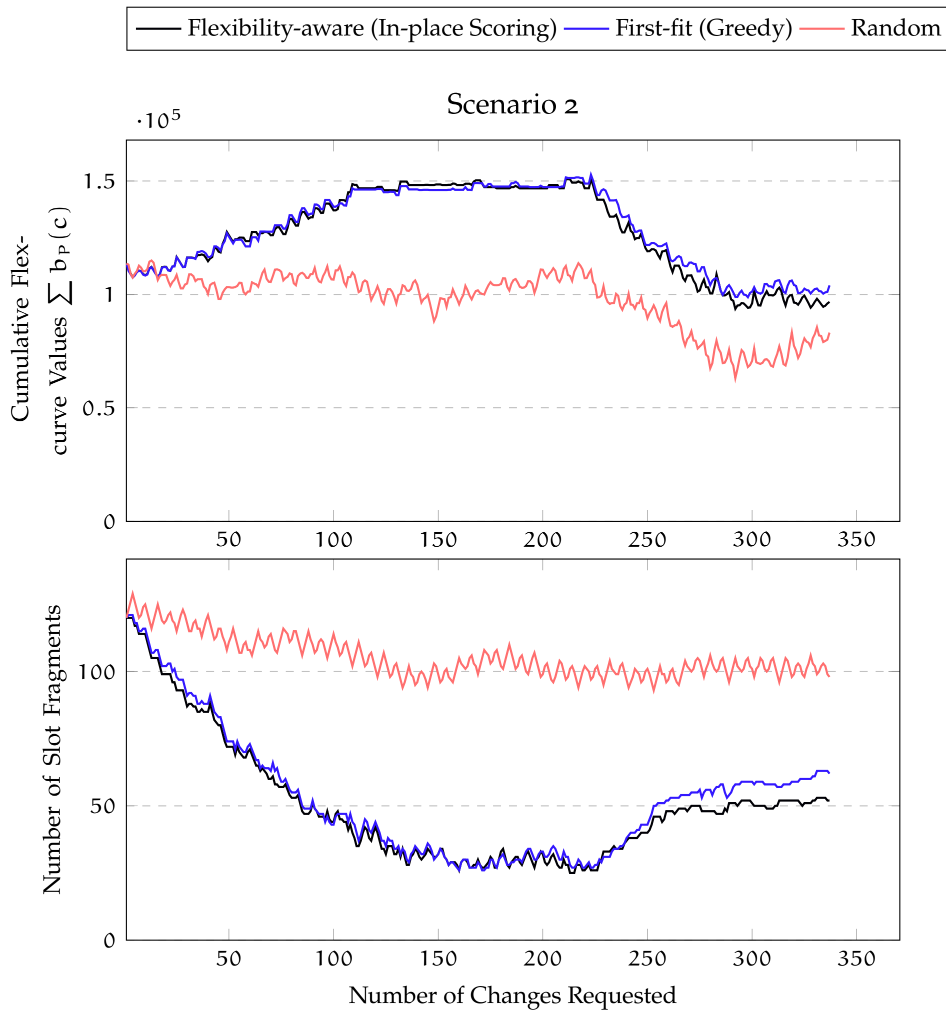


Figure 5.11: Flow requests with scenario 2 (Appendix A.4). Figure derived from [27].

6 MULTI-MECHANISMS

In this chapter, the scope of this thesis is expanded beyond Time-Sensitive Networking (TSN) and scheduled traffic (ST). Research goal 2 is addressed by introducing two methods to enhance TSN deployment capabilities. In Section 6.1, we describe a method for deploying TSN scheduled traffic using programmable push-in-first-out (PIFO) queues. In Section 6.2, we present the novel residence delay aggregation (RDA) method. RDA can be implemented on programmable network switches while ensuring per-flow delay guarantees. Finally, in Section 6.3, the concept of the flexcurve is extended to support Credit-based Shaper (CBS) flows. It is essential for the flexcurve metric to incorporate CBS to facilitate the deployment of multiple mechanisms on the same egress port using the flexcurve and scheduled traffic.

6.1 PIFO STRUCTURES IN TSN

The Section 6.1 is taken verbatim from [29]. The notation is adapted to align with the overall thesis notation.

Time-Sensitive Networking (TSN) has risen in recent years as an approach to convergent real-time networking standard with deterministic guarantees for industrial applications. Currently, TSN relies on special switching hardware which deterministic ensures latency guarantees for a handful of given scheduling mechanisms. One popular mechanism for scheduling real-time flows is the Time Aware Shaper (TAS), standardized in IEEE 802.1Qbv. It allows the programming of cyclic open and close instructions regulating queues with strict priority transmission selection at supported switch output ports. These instructions allow providing so called *scheduled traffic*, i.e. the real-time traffic class, with predefined transmission windows for jitter and loss free communication.

Push-In-First-Out (PIFO) [89] queues can be regarded as a priority queuing concept, designed for line-rate deployability in hardware. Packets can be inserted at an arbitrary position in the queue, but are always dequeued from the head. Enqueueing a packet at a certain position corresponds to the *rank* of that packet relative to the enqueued packets. This versatile concept allows expressing different types of schedulers such as priority and Least Slack Time First schedulers [89]. Current trends allow anticipating upcoming off-the-shelf switching hardware with PIFO support, e.g. as an extension to P4 data-plane programmable switches.

In this work, we propose utilizing PIFO queues to express the main functionality of the TSN Time Aware Shaper mechanism. We show how it can be used to realize non-overlapping scheduled traffic. Using a queuing model like PIFO requires the computation of a rank at the time of enqueue, and in addition, the design of an appropriate hier-

archical queue structure, such that the desired scheduling algorithm can be mapped to a PIFO structure. Note that the PIFO queuing concept supports hierarchical queueing, which are drained from the root [89].

Scheduled Traffic using PIFO queues

The Time Aware Shaper has a port cycle time, and up to eight priority queues, that can be opened and closed subject to hardware-specific time-granularity. A frame is not transmitted out of a queue if there is not enough time available until the next gate close instruction.

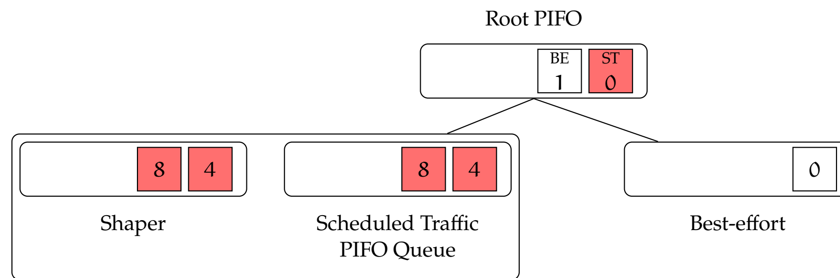


Figure 6.1: Hierarchical PIFO queue structure to support scheduled traffic (ST) windows and best-effort (BE) traffic. Each queue-element is sorted by its rank. The root references PIFO queues of the leaf. The shaper is responsible for enqueuing ST references when the clock reaches the assigned rank. Figure derived from [29].

We consider two classes of traffic that are supported by our approach: (a) scheduled traffic (ST), which has cyclic windows designated for pre-computed real-time flows, and (b) best-effort (BE) traffic, which does not receive service guarantees. Scheduled traffic windows are provided by scheduling algorithms such as [18]. Using a PIFO queue hierarchy as depicted in Fig. 6.1, we can ensure that scheduled traffic packets are transmitted in their designated time-slots. This is enabled by a secondary shaping PIFO queue, which holds back the enqueuing of references to the scheduled traffic queue into the root PIFO until the time of their designated window is reached. The scheduled time at the initial enqueue of a scheduled packet also directly gives the packet rank, i.e. its order.

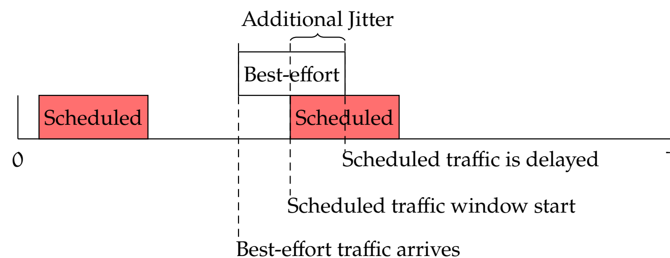


Figure 6.2: BE packets scheduled using PIFO just before scheduled traffic windows can result in additional jitter per hop. Figure derived from [29].

A naive approach is to let the root PIFO queue prioritize all scheduled traffic and de-prioritizes all best-effort traffic by assigning a rank of 1 for best-effort and 0 for scheduled traffic at the root PIFO. However, applying this approach would introduce jitter for scheduled traffic windows. This occurs when the transmission time of best-effort traffic overlaps with the reserved window of scheduled traffic (cf. Figure 6.2), and may repeat if multiple packets are transmitted within the reserved window. This per-hop jitter is bounded by the transmission time of the best-effort packet MTU, e.g. 12µs at 1 Gbps with 1514 byte packets. In standard TAS this behavior is avoided as the gate responsible for best-effort traffic can be closed when a scheduled traffic gate is open. Furthermore, TAS switches use implicit or explicit guard-bands to ensure no BE frame overlaps onto scheduled traffic windows.

To ensure this behavior with our approach we need to provide the guard-band functionality using the PIFO concept. Since we cannot delay the BE queue like a standard TAS, BE packets need to be scheduled dynamically based on the reserved scheduled traffic windows at the corresponding port. This must be done at each hop along a network path of a stream. Note that dynamical scheduling of BE packets in the proposed approach is not trivial, since loops are not directly supported by programmable switching hardware [14]. Hence, we cannot shift the scheduled time of new BE packets until the scheduled time is cleared of the reserved window.

Our approach relies on a look-up table (LUT), as depicted in Figure 6.3, to provide this behavior. By segmenting the port cycle into scheduled and non-scheduled windows, we can lookup the required information to avoid overlaps of BE packets onto scheduled traffic. For each window we keep a vector with *(i)* the number of time-units until the next reserved scheduled traffic window, *(ii)* the next scheduled traffic window-size, as well as *(iii)* the beginning time of the current window. All time-points are relative to the start of the port cycle time. The sum of a window's vector will correspond to the next *possible insertion* position relative to the beginning of a cycle. Therefore, the next **`window-size`** must point to a time-point, where at least one MTU-sized best-effort packet can be scheduled without overlapping onto a reserved window, i.e. reserved window gaps smaller than one MTU are ignored. The rank calculation of the best-effort shaper is sketched in Algorithm 4, with **`last_BE_endtime`** representing the time point, at which the transmission of the last scheduled best-effort packet ends. This and the rank are the only modified state.

A LUT can be implemented within a programmable switch using match-action units. The contained tables support range matches, which in turn support the retrieval of the window-vector. In case no range matches are supported by the device, they can be realized by multiple prefix or exact matches.

The approach illustrated here is comparable to using implicit guard-bands. A best-effort packet is sent before reserved scheduled traffic windows, if the packet size permits. However, if there are gaps of less than one MTU, this approach cannot populate these gaps, due to the safety margin within the window-size variable. To support this, it would require new entries in the LUT for different possible BE packet sizes.

Range Match

0 1 2 3 4 5 6 7 8 9 10 11

ST ST

Free Slots	1	0	4	0	3
Begin	0	1	3	7	9
Window Size	2	2	2	2	2

Figure 6.3: Look-up table encodes the number of time-units until the next reserved window, as well as the size of the next reserved window. When used with range-lookups the table must also reference the beginning of the matched range. Figure derived from [29].

Algorithm 4 : Rank for Best-Effort PIFO Queue Shaper. Derived from [29].

```
   Data : Packet p(frameduration);
   last_BE_endtime;
1  if last_BE_endtime < NOW then
2  |  last_BE_endtime = NOW;
3  rel_pos_start = last_BE_endtime % CYCLE_TIME;
4  lut = LUT(rel_pos_start);
5  slots_available = lut.begin - rel_pos_start + lut.free_slots;
6  if slots_available < p.frameduration then
7  |  p.rank = last_BE_endtime - rel_pos_start + ∑_i lut.i;
8  else
9  |  p.rank = last_BE_endtime;
10 last_BE_endtime = p.rank + p.frameduration;
```

Conclusion

In this work we presented a novel approach for replacing special time-sensitive networking hardware by programmable of-the-shelf switches with PIFO queues. We showed how to use PIFO queues to schedule real-time TSN traffic together with best-effort traffic. We provided an algorithm that can be directly implemented on programmable switches with a PIFO programmable traffic manager to isolate scheduled traffic from best-effort traffic using a guard-band functionality.

6.2 RDA: RESIDENCE DELAY AGGREGATION

The Section 6.2 is taken verbatim from [100]. The notation is adapted to align with the overall thesis notation.

Time-Sensitive Networking (TSN) is a collection of standard extensions for the IEEE 802.1Q [42] Ethernet standard. TSN introduces multiple mechanisms that enable guaranteed Quality of Service (QoS) based on standard Ethernet. Different TSN mechanisms are able to offer different kinds of guarantees, ranging from token bucket traffic shaping to deterministic scheduled traffic with zero jitter. Without TSN mechanisms, the sole use of basic static priority mechanisms is known to suffer from starvation problems for the best-effort traffic class. For example, the Credit-based Shaper (CBS) restricts the flow rate, and the Time Aware Shaper (TAS) may close higher priority queues for a certain time period [23].

TSN mechanisms are deployed on network switches, where scheduling and shaping decisions are made according to the used QoS mechanism and its configuration for egress ports. To integrate known TSN schedulers into a switch, the egress port scheduler needs to be capable of the intended mechanism at the design phase or capable of such a configuration later.

The domain-specific language P4 [14] enables the programmability of the data plane behavior on nowadays available network devices. However, P4 is constrained in its ability to program the behavior of a device's egress queue scheduling. Nevertheless, P4 devices can be programmed to run or emulate some stateful scheduling techniques, including variations of the priority queuing PIFO-mechanism [3] or Active Queue Management (AQM) algorithms [58].

Problem Statement

Typically, TSN configuration deployments are considered static. The network is not dynamically adjusting to current load situations. While critical traffic flow behavior and requirements may change over time, the deployed configuration does not adapt automatically. Furthermore, mechanisms with static resource reservations waste resources unnecessarily when flows change dynamically. We seek to find a resource-sharing mechanism that can be deployed on off-the-shelf programmable network devices like P4 switches.

In this paper, we analyze the Residence Delay Aggregation (RDA) approach, specifically designed for implementation on P4 switches. Unlike the traditional Least Slack Time First (LSTF) scheduling strategy, which prioritizes packets based on the remaining time to their deadline and necessitates a priority queue, RDA provides a solution for real-time dynamic scheduling on current P4 switches. Importantly, RDA dynamically determines when critical traffic should be prioritized.

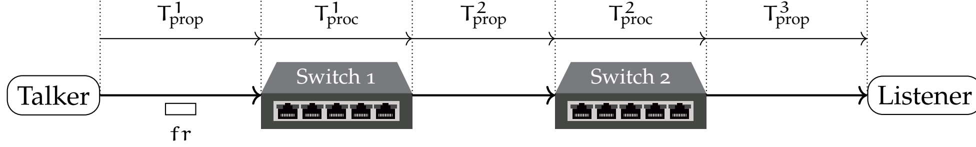


Figure 6.4: Time-consumption along the transmission path. T_{prop} indicates propagation delay. T_{proc} indicates processing time within the switch. Figure derived from [100]

Residence Delay Aggregation

The core concept of RDA, similar to LSTF, is that the priorities of deadline-carrying packets are determined by the total residence delay of packets as they travel through the network. The residence delay represents the cumulative time that a packet spends on switches while traversing the network. In a nutshell, packets with deadlines specified by the application are given a time allowance, encoded in the header of each packet, during which they can traverse between switches in best-effort queues (BEQ) without special handling. Upon the arrival of a deadline-carrying packet in a switch, the allowance time is checked before enqueuing the packet. If the allowance time does not suffice to meet the packet's deadline, the packet will instead be enqueued at a high-priority urgent queue (UQ).

Network Delays

TSN mechanisms achieve deterministic communication by providing upper bounds on the end-to-end delay. Figure 6.4 illustrates a simple network with two P4 programmable switches. The sender generates a packet (fr) which is delivered to the receiver through switches sw_1 and sw_2. In addition to the link propagation delay (T_{prop}), each switch comprises a processing delay (T_{proc}) which we describe here as

$$T_{proc} = t_{parser} + t_{ingress} + t_q + t_{egress} + t_{deparser} \tag{6.1}$$

In this paper, we let T_{proc} comprise the internal operations taking place within the programmable switches such as queuing and pipeline operations, as illustrated in Figure 6.5. Further elaboration of this figure is presented in Section 6.2.

Upon the arrival of a packet in a switch, it is initially parsed by the parser and subsequently proceeds through the ingress pipeline until reaching the traffic manager. The time consumed by parsing and ingress processing is denoted by t_{parser} and $t_{ingress}$, respectively. The packet is then enqueued in the traffic manager and waiting to be dequeued. This queuing delay, denoted by t_q, depends on the total amount of traffic pushed in the queues and on the applied traffic shaping and scheduling algorithms. After the packet is dequeued from the traffic manager, it proceeds through the egress pipeline (assuming the switch offers an egress pipeline for packet processing) and the deparser to reach the egress port. The time consumed by the egress pipeline and deparser is denoted by t_{egress} and $t_{deparser}$, respectively. We assume that the factors t_{parser}, $t_{ingress}$, t_{egress}, and $t_{deparser}$ are almost constant. Therefore, t_q stands as the sole factor that can affect T_{proc} and, consequently, impact the overall end-to-end latency.

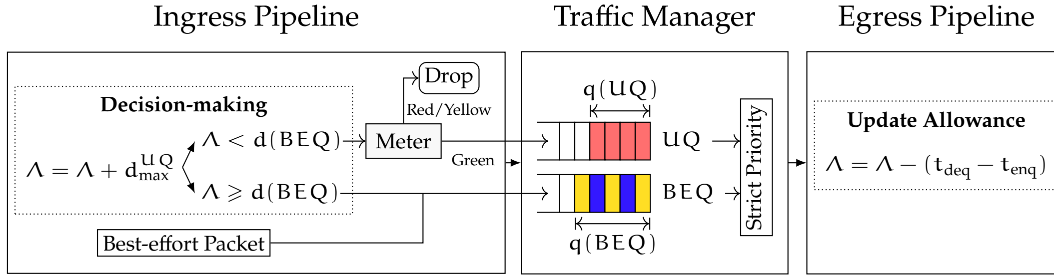


Figure 6.5: System design of RDA. Packets in traffic manager: Red ■: urgent traffic, Yellow ■: non-urgent traffic, Blue ■: best-effort traffic. t_{enq}: timestamp when packet is enqueued. t_{deq}: timestamp when packet is dequeued. Figure derived from [100]

Queuing Model

The existing TSN mechanisms prioritize traffic based on their types. According to the Class of Service in IEEE 802.1 Q standard, a 3-bit field Priority Code Point is presented in the VLAN header to indicate eight different classes of traffic. Therefore, a minimum of eight queues associated with fixed priority levels for each egress port are needed to isolate distinct traffic if the switch is required to support the complete set of traffic classes. Strict priority transmission selection ensures that higher-priority traffic is dequeued before lower-priority traffic, hence, the time-critical sessions running on the end devices can be accomplished in a timely manner.

One goal of TSN is to guarantee that packets reach their destination before a deadline. The urgency of reaching the deadline is the most critical factor for prioritizing traffic. The urgent traffic, which has an imminent deadline, should be assigned with higher priority than the non-urgent traffic, which still has a loose gap until the deadline.

In RDA, each switch dynamically sets traffic priorities based on two factors: the remaining time to meet the deadline and the current fill level of queues in the switch. Traffic may become more or less urgent in subsequent switches, depending on the queuing delay in the current switch. This dynamic priority assignment enables balancing traffic load while providing time guarantees for deadline-carrying packets.

RDA uses two separate queues for each egress port to buffer three types of traffic, as shown in the traffic manager in Figure 6.5. An urgent queue (UQ) assigned with high priority is designated for buffering the urgent traffic, whereas a best-effort queue (BEQ) assigned with low priority is used for buffering the best-effort and non-urgent traffic. With strict priority transmission selection, the urgent traffic in the UQ is ensured to be transmitted before any traffic in the BEQ. A meter (also called policer) is applied for UQ to limit the urgent traffic rate. The meter has the burst and rate parameters (B_{meter}, r_{meter}) that denote the maximum burst size and the maximum long-term rate allowed for the UQ flows, respectively [60].

Note that, similar to existing TSN mechanisms, the deadline for packets can be guaranteed when the packet passes through the UQs of all switches along the path. In order to determine whether a packet is urgent or non-urgent, the switch should answer the question: Can the deadline of the incoming packet still be guaranteed when buffering

the packet in the BEQ of this switch and in the UQs of all subsequent switches? Therefore, RDA prescribes that the packet header of each deadline-carrying packet contains a field called *time allowance* denoted as Λ, which specifies how long this packet can be buffered in the BEQ over the transmission path for switches $\omega \in \{1, ..., m\}$. The i-th switch is denoted by sw_ω. The sender sets the initial value of Λ as

$$\Lambda = d - \sum_{l} T^{l}_{prop} - \sum_{\omega} T^{\omega}_{pipe} - \sum_{\omega} t^{\omega}_{UQ,max} \quad (6.2)$$

where d is the deadline for reaching the destination. Here, l indicates the index of links along the transmission path; t^{ω}_{pipe} represents the time consumed in sw_ω except for the queuing delay in the traffic manager. The variable

$$t^{\omega}_{UQ,max} = \frac{B^{\omega}_{UQ,meter}}{r^{\omega}_{line}} \quad (6.3)$$

for sw_ω indicates the maximum queuing delay for the UQ, i.e., the maximum time until packets are served when assigned to this queue. $B^{\omega}_{UQ,meter}$ is the burst size for the meter assigned to the UQ in sw_ω. The line rate for the egress port in sw_ω is given by r^{ω}_{line}. The maximum queuing delay for transmitting packets exclusively through the UQ in all switches is denoted as $\sum_{\omega} t^{\omega}_{UQ,max}$. This value, propagation and pipeline delays are then subtracted from the given deadline to establish the allowance time Λ for buffering in the BEQ.

The initial value of Λ is given by the sender, or the network controller which needs awareness of the delay values (6.2) along the path. Λ is initialized to be $\geqslant 0$ to ensure on-time delivery because worst-case delays for subsequent UQs are accounted for during the initialization of Λ. Specifically, if no time remains for enqueuing the packet into BEQs, UQs are exclusively used to ensure an on-time packet delivery.

Note that, first, this proposal is conservative in the sense that we assume worst-case delays at every UQ. Secondly, with this method, we need to adjust Λ with the actual queuing delay at each switch after the packet is dequeued to reflect the actual residence time. For an initial value of $\Lambda < 0$, the packet still may arrive within its deadline due to not fully utilized UQs along the path; however, no guarantees can be given.

Figure 6.5 depicts the control flow when a packet (fr) enters the switch. If fr is a deadline-carrying packet observed by the parser, the decision-making process in the ingress pipeline is triggered. The value of Λ is first increased by $t^{\omega}_{UQ,max}$ of this switch sw_ω. This adjustment allows reclaiming the subtracted maximum queuing delay in the UQ from the packet deadline, as the packet has not yet entered the UQ at sw_ω. This ensures the accurate representation of the packet's deadline constraints as it moves through the network.

The switch then compares the value of Λ with the threshold $t^{\omega}(BEQ)$ to decide which queue is eligible for fr. The value of $t^{\omega}(BEQ)$ represents the queuing delay if fr is buffered into the BEQ. This delay encompasses the transmission time of the current packets in the BEQ and UQ, as well as the upper bound delay caused by any packets

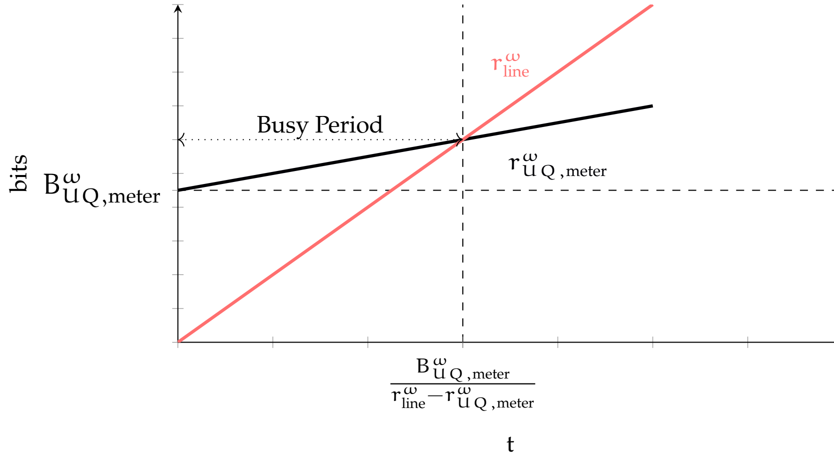


Figure 6.6: The arrival and service curves of UQ. The arrival curve is colored in black, while the service curve is in red. Figure derived from [100]

that will enter the UQ in the future before fr is eventually dequeued. Therefore, the classical delay bound $t^{\omega}(BEQ)$ at sw_{ω} is given by

$$t^{\omega}(BEQ) = \frac{q^{\omega}(BEQ)}{r^{\omega}_{line} - r^{\omega}_{UQ,meter}} + \frac{B^{\omega}_{UQ,meter}}{r^{\omega}_{line} - r^{\omega}_{UQ,meter}} \tag{6.4}$$

where $q^{\omega}(BEQ)$ is the current depth of the BEQ, observed by an arriving packet. Should a packet be in risk of being dropped while being enqueued into the BEQ when BEQ is full, $t^{\omega}(BEQ)$ is assigned a value of ∞, to force the deadline-carrying packet into the UQ instead. The second term denotes an upper bound on the busy period of the UQ. This is a worst-case bound, as it assumes that the urgent packets are always enqueued into UQ at the meter rate ($r^{\omega}_{UQ,meter}$). The value of this upper bound is determined leveraging network calculus theory [60], as shown in Figure 6.6. The arrival curve for UQ is expressed by affine function $\gamma_{r,b}(t) = rt + b$, where $r = r^{\omega}_{UQ,meter}$ and $b = B^{\omega}_{UQ,meter}$. The service curve for UQ is represented by peak rate function $\lambda_R(t) = Rt$, where $R = r_{line}$. The busy period of UQ lasts until the backlog reaches 0, which is the value of the second term in (6.4).

If $\Lambda \geqslant t^{\omega}(BEQ)$, then the packet is eligible to be enqueued in the BEQ, as the expected waiting time would not exceed the packet's deadline at sw_{ω}. Otherwise, the UQ is selected. Recall that the UQ should be metered to ensure an upper bound of $t^{\omega}(BEQ)$. The packet fr is dropped if it is set to be enqueued into UQ and UQ is currently full. This dropping mechanism ensures that the packets transferred to UQ see at most the delay described in (6.3); however, such packet dropping should never appear under normal circumstances. The meter can also be applied to different traffic sources (e.g., based on flows) to prevent UQ from being flooded by a specific source with urgent traffic. Finally, after being dequeued, the value of Λ in the packet header is decreased with the **residence delay** ($t^{\omega}_{deq} - t^{\omega}_{enq}$) in the Egress pipeline of sw_{ω}. The

updated value of Λ is utilized at the next switch to decide which queue would be used for buffering this packet.

Limitations of Programmable Switches

The computation of $t^{\omega}(\text{BEQ})$ requires knowing the current depth of the BEQ in the ingress pipeline, which is not supported by all programmable switch architectures (e.g., Tofino). The Tofino2 switch architecture allows a ghost thread to retrieve queue depths from the traffic manager to the ingress pipeline [2]. To accommodate different programmable switch architectures, the threshold $t^{\omega}(\text{BEQ}) = t^{\omega}_{\text{BEQ,max}}$ (compare (6.4)) can be modified to the static

$$t^{\omega}_{\text{BEQ,max}} = \frac{q^{\omega}_{\text{BEQ,max}} + B^{\omega}_{\text{UQ,meter}}}{r^{\omega}_{\text{line}} - r^{\omega}_{\text{UQ,meter}}} \tag{6.5}$$

where $q^{\omega}_{\text{BEQ, max}}$ is the maximum depth of the BEQ at sw_{ω}.

Note that the P4 language lacks support for division operations [58]. Hence, one approach to approximate the division operation is through performing bit shifting which requires the denominator to be a power of 2 for exact results [32]. To this end, the meter rate needs to be configured to a value that ensures $(r^{\omega}_{\text{line}} - r^{\omega}_{\text{UQ,meter}})$ is a power of 2.

Implement RDA in P4 Programmable Switches

The implementation of RDA relies on the programmable switch architecture, as described in Section 6.2. The pseudocodes interpreted in this subsection are based on the bmv2 switch, which is a programmable software switch and does not support retrieving the current queue depth in the ingress pipeline. Therefore, equation (6.5) is leveraged as the delay threshold for the decision-making procedure.

The P4 pseudocode for the ingress processing is presented in Listing 6.1. The values of $t^{\omega}_{\text{UQ,max}}$, $q^{\omega}_{\text{BEQ,max}}$, $B^{\omega}_{\text{UQ,meter}}$, and the exponent n in (6.6) are stored in the metadata of the programmable switch by **`uq_max_del`**, **`beq_max_dep`**, **`uq_bs`**, and **`rate_diff_power`**, respectively. These values are configured by the controller through the match-action table. Upon the arrival of a deadline-carrying packet, which is determined by the tag **`is_deadline_packet`**, the meter for constraining the rate of entering UQ is triggered. Note, that there is only one meter applied to UQ itself, not to each individual flow. The execution of meter results in three colors, as defined in RFC 2697 [35]. The **`meter_tag`** stores the result of the meter execution (according to the meter rate $r^{\omega}_{\text{UQ,meter}}$). A green result is indicated by 0. Once the deadline-carrying packet is allowed to be enqueued in UQ, the value of $t^{\omega}_{\text{UQ,max}}$ is reclaimed to the allowance time carried in the packet header. The value of delay threshold $t^{\omega}_{\text{BEQ,max}}$ is computed by shifting **`q_dep`** to the right by n bits, computed in:

$$2^{n} = r^{\omega}_{\text{line}} - r^{\omega}_{\text{UQ,meter}} \tag{6.6}$$

The arrived deadline-carrying packet is assigned a queue ID of 1 if the allowance time is less than the delay threshold. A higher queue ID corresponds to a higher queue

Listing 6.1: P4 pseudocode for decision-making, meter usage and queue assignment. Listing derived from [100]

```
1  if (meta.is_deadline_packet == 1) {
2      meter_uq.execute_meter(0, meta.meter_tag);
3      if (meta.meter_tag == 0) {
4          hdr.rda.at = hdr.rda.at + meta.uq_max_del;
5          q_dep = meta.beq_max_dep + meta.uq_bs;
6          del_thr = q_dep >> meta.rate_diff_power;
7          if (hdr.rda.at < del_thr) {
8              standard_metadata.priority = 1;
9          } else {
10             standard_metadata.priority = 0;
11         }
12     } else {
13         drop();
14     }
15 } else {
16     standard_metadata.priority = 0;
17 }
```

Listing 6.2: P4 pseudocode for updating the allowance time. Listing derived from [100]

```
1  if (meta.is_deadline_packet == 1) {
2      hdr.rda.at = hdr.rda.at - standard_metadata.deq_timedelta;
3  }
```

priority. Thus, the queue with ID 1 indicates the UQ, whereas the BEQ is represented by the queue with ID 0.

Updating the allowance time in the egress pipeline is shown in Listing 6.2. The field **deq_timedelta** in **standard_metadata** stores the value of the time consumed by queuing in the traffic manager, which is then deducted from the current value of the allowance time.

Related Work

We briefly discuss related concepts in the following. The Push-In-First-Out (PIFO) [89] priority queuing concept is a promising approach to enable a flexible configuration of egress port schedulers. It is capable of emulating schedulers like Earliest Deadline First (EDF), Least Slack Time First (LSTF), and TSN Time Aware Shaper [29]. However, support for PIFO is unavailable on off-the-shelf devices and FPGAs are needed. Similarly, for other queueing challenges the use of FPGAs has proven to be a practical solution [57].

We also want to highlight one particular TSN mechanism due to similar goals: Asynchronous Traffic Shaping (ATS) as given by [43, 91]. Both RDA and ATS aim to provide end-to-end delay bounds per flow without reservations but achieve this through differ-

ing mechanisms. ATS employs multiple queues for each egress port to separate ingress flows based on ingress port and priority. Each queue calculates an eligibility time for its topmost flow, retaining packets until this time is met. Unlike RDA, there is no need for additional encoded information (allowance time) in packets during transmission. However, ATS necessitates a more complex hardware support including queuing structure and queuing scheduling, leading to increased resource overhead compared to RDA. Currently, no off-the-shelf devices are available that support ATS with the required queuing and scheduling strategies. In contrast, RDA only requires priority queues and a strict priority transmission selection algorithm, which is specified as the default algorithm for selecting frame for transmission in IEEE 802.1Q [42]. RDA is implementable on Tofino-based switches solely with the availability of strict priority scheduling and FIFO-queues.

Conclusion

We proposed a new dynamic scheduling mechanism called Residence Delay Aggregation (RDA), which is designed to provide time guarantees for time-critical traffic while only utilizing priority queuing when necessary to deliver packets timely. The presented approach is built on top of dynamic traffic scheduling at each switch, prioritizing traffic based on the urgency of meeting packet deadlines and the current queue states. RDA is designed to be implementable on the modern P4 programmable switches. In future work, we plan to assess the performance of RDA across various hardware platforms in different network topologies and traffic scenarios. The impact on the performance due to the limitation imposed by the P4 programming language and the underlying concepts will be investigated. Additionally, we will extend RDA by incorporating multiple urgent queues to enhance its flexibility in prioritization.

6.3 FLEXIBILITY OF SIMULTANEOUS USAGE OF TSN MECHANISMS

In the preceding Chapters 4 and 5, we introduced the concept of the flexcurve and applied it to TSN scheduled traffic (ST). When traffic with less stringent requirements – which can be satisfied by mechanisms other than TSN ST – is requested, it is appropriate to select mechanisms suitable for this type of traffic. Choosing the appropriate mechanism can reduce unnecessary reservations of network resources. For instance, unlike ST, asynchronous mechanisms do not explicitly reserve time slots. The separation and simultaneous handling of different traffic types can be achieved by reserving distinct queues for each type at the corresponding egress port.

The flexcurve does not consider the impacts of other types of real-time traffic that are also currently active simultaneously in the network. A potential scenario involves the usage of ST alongside TSN Credit-based Shaper (CBS) flows. The CBS enables the calculation of delay bounds [20, 64, 67, 68, 70, 73, 98, 99] as well, is however unsuitable to support the more stringent isochronous flows. When separating the traffic types by queue, operating traffic that has less stringent requirements compared to those enabled by ST, is enabled.

The simultaneous deployment of scheduled traffic and CBS impacts the observed delay of CBS queues by essentially pausing the credit mechanism of the CBS (cf. Section 2.2), when the higher priority isochronous traffic is scheduled. To integrate awareness of additional real-time traffic types, particularly CBS, we propose an extension of the flexcurve. This extension is akin to the integration of deadline awareness, in that it requires additional parameters to be provided, which are respected by the resulting flexcurve values. The deadline-aware flexcurve $b_P^d(c)$ (cf. Sections 4.1.3 and 5.1) is based on an eligibility list $\overline{\mathcal{A}}$, that includes assignments A for which the given parameters of path P, frame size c, and deadline d are respected. Similarly, we argue that a flexcurve, which is CBS-aware, needs to respect the requirements of such CBS traffic.

The CBS traffic properties that need to be respected include the deadlines and transmission eligibility durations per CBS class per port. Transmission eligibility durations are determined by the expected maximum delay, combined with the maximum burst duration traffic is allowed to transmit at the port. The CBS deadlines per port p along path $P \in \{p_1, \dots, p_m\}$ per CBS class $w \in \{1, \dots, 8\}$ are given by cd_p^w, along with their corresponding eligibility durations ed_p^w.

When populating the eligibility list $\overline{\overline{\mathcal{A}}}$, for example, using Algorithms 2 and 3, candidates can now only be considered eligible (in addition to the ST requirements for deadline d and frame size c), if the CBS eligibility durations ed_p^w do not exceed the corresponding CBS deadlines cd_p^w as a result of admitting the candidate. The deadline can be exceeded, because placements of ST pause the transmission of CBS queues.

Quick Eligibility Check

We can check this condition quickly by evaluating the total number of ST slots, determined either by the sum of all frame sizes in the port schedule or by the residual capacity using $h - b_p(1)$. However, this condition is merely sufficient; there may be too many total reserved slots. The transmission of CBS frames is asynchronous, and

hence, can occur at arbitrary points in time. Therefore, in scenarios where the simple frame size approximation is insufficient, it becomes necessary to conduct a worst-case transmission time search for the CBS-traffic type. This determines whether the potential begin of a CBS transmission, coupled with the admission of a candidate A and the remaining schedule entries, might surpass the CBS deadline at the port at fragmented schedules.

Worst-case Transmission Time

A worst-case transmission time is given when the maximum number of ST reserved slots pause the credit mechanism of the CBS. We visualize this search in Figure 6.7. Reserved slots, including the candidate A, are colored red. In the example, the maximum extension of the eligibility window for the first CBS queue is given at starting time 1. This worst-case placement results in a worst-case eligibility duration of $ed_p^1 + 2$. However, this still results in fewer slots than the corresponding CBS deadline cd_p^1 at the port. Therefore, with the current state of the port schedule, the candidate, if any, can be deemed eligible.

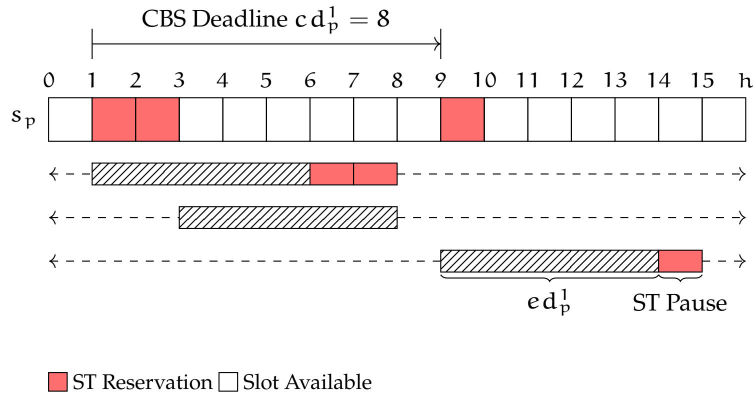


Figure 6.7: Example of a worst-case placement for a CBS transmission window. The worst-case placement is achieved when the maximum number of reserved slots interfere with the CBS transmission eligibility window.

Worst-case Transmission Time Search

A fast way of finding the worst-case placement for a CBS transmission window is to use a dynamic sliding window algorithm. For a given port schedule s_p, and CBS eligibility duration $\mathbf{ed} := ed_p^w$ for the class w, we depict a possible approach in Algorithm 5. Note, that a potential candidate A needs to be included in the searched schedule, as if an admission was performed.

The algorithm can be implemented linearly by iterating through each frame transmission start s_p^i within the port schedule s_p, assuming starting times are sorted: $s_p^1 < s_p^2 < s_p^{\chi_p}$. This narrows the number of slots needing to be checked significantly from hyperperiod h-slots to χ_p-schedule entries. It is sufficient to check only these slots, as with each starting time of a frame s_p^i, the complete ST pause of this instance will

be regarded. Other starting times cannot constitute a worst-case, as less interruption is given by the ST frame. This way, there are no partial overlaps, as the subsequent schedule entries also participate in their full duration to the ST pause, should they overlap with the growing window.

In a nutshell, the worst-case search of Algorithm 5 iterates through each schedule entry s_p^i, increasing the eligibility duration window **ed** with each schedule entry's frame duration where the window overlaps. Each entry is a potential starting position for the worst case. The window is moved to a new potential starting position when an entry does not overlap with the previous window anymore.

In more detail, Algorithm 5 requires the following variables to keep track of the current window and worst-case time (Lines 1-4):

- **potstart**, reflecting the current search window's starting time.
- **ext**, reflecting the number of slots (duration) that are extended by all overlapping schedule entries. The search window's length is a result of the eligibility duration **ed** plus the window extension **ext**.
- **wcext**, **wctime** are keeping track of the current worst-case values for the extension **ext** and potential start **potstart**.

The algorithm begins by iterating through each schedule entry (Line 6). Each schedule entry extends the duration of the window (Line 14). The worst-case time and extension are updated in Lines 16-18. The window is extended until the iterated port schedule entry i does not overlap with the current extended eligibility window (Line 8). If the extension does not overlap, the search window start **potstart** is shifted to the new entry's position (Line 10). The current window extension is reset as we move to a new potential starting position (Line 12).

The algorithm returns the worst-case starting time **wctime** and window extension duration **wcext** (Line 19).

To reflect wraps, we can extend the schedule. The schedule is extended for each entry by shifting the transmission time by the hyperperiod:

$$(s_p^1, \ldots, s_p^{\chi_p}, s_p^{\chi_p+1} + h, \ldots, s_p^{2\chi_p} + h)$$

This process essentially involves copying each entry and appending the entries one cycle later. Starting times need to be checked only until the potential window start **potstart**, either exceeds or matches the hyperperiod: **potstart** $\geqslant h$.

The variable **wcext** tracks the current worst-case extension of the CBS window. If the worst-case extension **wcext**, does not exceed the corresponding CBS deadline cd_p^w, the checked schedule can accommodate the CBS class. This implies that the current scheduled traffic does not interfere in a way that prevents the CBS requirements from being met. Consequently, if a candidate A is included in the port schedule s_p, the candidate can be deemed eligible with respect to the CBS class constraints.

Integrating this worst-case search within Algorithms 2 and 3 prior to "saving" the candidate A as part of the eligibility list ensures that the CBS-traffic requirements are satisfied. This modification substantially expands the parameter set of the flexcurve by explicitly accounting for other traffic types.

Concerning an integration without deadline-awareness: the use of the eligibility list enables the rejection of candidates based on supplementary requirements. This capability is absent in the basic flexcurve formulation Eq. (4.2), where candidates for eligibility are not explicitly selected.

Algorithm 5 : Find the CBS worst-case starting time for a given port.

Input :

- The port schedule s_p with χ_p-entries for which the CBS worst-case starting time should be searched. The candidate A is assumed to be integrated in this given schedule. The schedule is assumed to be sorted: $s_p^1 < s_p^2 < s_p^{\chi_p}$.
- The duration(s_p^i) of occupancy at port p for a frame instance s_p^i.
- The CBS eligibility duration which is checked: **ed**

  Note: We omit a second pass with shifted schedule to consider schedule wraps and early returns from empty schedules.

Result :

- **wctime**: The worst-case starting time
- **wcext**: The overlap duration for *wct*

```
1  ext ← 0 // Current window extension (total ST pause duration)
2  potstart ← s_p^1 // Potential starting time, initialize to first entry
3  wcext ← 0 // Worst case ST pause
4  wctime ← 0 // Worst case starting time
5  // Iterate schedule entries i ∈ {1, ..., χ_p}:
6  for i in 1, ..., χ_p do
7      // Check if interval [potstart, potstart + ed + ext] does not overlap
        with current frame instance s_p^i
8      if s_p^i ∉ [potstart, potstart + ed + ext] then
9          // Move to next potential start
10         potstart ← s_p^i
11         // Reset search window extension
12         ext ← 0
13     // Update the window extension with corresponding frame duration
14     ext ← ext + duration(s_p^i)
15     // Update worst-case time and overlap duration
16     if ext > wcext then
17         wcext ← ext
18         wctime ← potstart
19 return wct, wco
```

7 SUMMARY, CONCLUSIONS, AND OUTLOOK

In this work, we have discussed various challenges associated with enhancing flexibility in Time-Sensitive Networking (TSN) and proposed approaches to address these challenges. To conclude, we summarize the preceding chapters and outline the main contributions. Finally, we explore avenues for future research.

7.1 SUMMARY OF THE THESIS

We motivated this work in Chapter 1 INTRODUCTION and discussed three essential challenges that hinder an inherent flexibility in TSN. As network applications evolve and requirements shift, such as manufacturing systems in a factory environment, there is a necessity for adaptation within network configurations and infrastructure. This becomes increasingly complex as applications demand strict traffic service requirements. We focused the main research direction of this work to scheduled traffic, which can support isochronous traffic types. In Chapter 2 BACKGROUND & RELATED WORK, we provided an overview of the relevant background information, and reviewed existing works and approaches related to this research. Consequently, we derived three specific research goals, aimed at enhancing flexibility:

- **Research Goal 1**: Provide a mechanism for analyzing flexibility of scheduled traffic configurations.
- **Research Goal 2**: Methods for optimizing the flexibility of TSN schedules.
- **Research Goal 3**: Enabling a flexible deployment of TSN mechanisms to achieve application requirements.

Taking into account these research goals, we summarize the contributions of this thesis as follows.

7.1.1 *Contributions*

The management process of TSN encompasses a variety of scenarios and numerous procedures. In Chapter 3 FLEXIBILITY-BASED TSN MANAGEMENT, we propose an enhancement for centralized TSN controllers to incorporate a flexibility metric, thereby significantly improving the adaptability of the scheduling process. This chapter provides context on the applicability of the contributions in this thesis. The metric may be consulted by the controller prior to making critical decisions. The metric serves as a tool to enhance decision-making processes across various domains. Its application is particularly crucial for determining the admissibility of new flows, ensuring the

network's capabilities can support the flow requirements without necessitating separate scheduling processes. By integrating the flexcurve into the scheduling process, a scheduler can steer the flow scheduling towards flexibility optimizations. Moreover, as a path-based metric, the flexcurve facilitates the selection of promising paths for future flows. Furthermore, we refine the parameters exchanged between the controller and applications to a selected set that we consider essential for integrating flexibility awareness into TSN management.

In Chapter 4 FLEXIBILITY NOTION, we address Research Goal 1, focusing on the conceptualization of flexibility within the context of TSN scheduled traffic. Our contribution, denoted *flexcurve*, quantifies the flexibility at the network's bottleneck by considering the flow frame size requirements, thus indicating possibilities for accommodating new flows. Contrary to most related works in this domain, we introduce a path-based metric, thereby capturing the intrinsic characteristics of scheduled traffic. This allows for a more detailed measurement of the network's environment. Moreover, we leverage the structure of the flexcurve to facilitate partial aggregations and disaggregations. Disaggregations permit simultaneous flow admissibility checks for multiple flows. This feature can assess the network's capability to accommodate additional traffic without requiring exhaustive, individual flow scheduling procedures. Aggregations, on the other hand, enable a rapid construction of the flexcurve for selected data points, by leveraging the underlying schedule's data-structure itself. Finally, we extend this flexibility notion further with additional deadline requirements that are specified by the user.

Building upon the foundational concepts introduced in the Chapter 4, Chapter 5 OPTIMIZATION addresses Research Goal 2 by detailing the integration of flexcurve-based scheduling into the scheduling process. We introduce a search heuristic algorithm for scheduling TSN scheduled traffic, which can also be utilized to identify flow scheduling candidates crucial for constructing a deadline-aware flexcurve. Through the introduction of two approximation methods for the pruning of eligibility candidates and in-place scoring during the algorithm's runtime, alongside a novel approach for path selection that considers potential flexcurve changes, we can efficiently leverage the flexcurve for flexibility-aware scheduling purposes. Finally, this chapter further extends our search algorithm to enable TSN gate control list (GCL) deployment. Supporting GCLs is critical for realizing actual TSN deployments of configurations on forwarding devices.

Continuing with Chapter 6 MULTI-MECHANISMS, we address Research Goal 3. Extending the variety of mechanisms provides more opportunities for deployment when different types of forwarding hardware can be leveraged. In Section 6.1, we facilitate the deployment of TSN scheduled traffic by proposing a configuration method for the push-in-first-out (PIFO) queueing concept. This adaptation allows TSN mechanisms to be deployed on hardware supporting this type of programmable scheduler. Secondly, in Section 6.2, we introduce a novel scheduling policy, denoted *residence delay aggregation (RDA)*. RDA is an asynchronous mechanism capable of ensuring per-flow delay guarantees on programmable hardware switches by utilizing two separate queues with simple strict priority for transmission selection. Finally, we incorporate

multi-mechanism support into the flexcurve metric for scheduled traffic by considering the requirements of Credit-based Shaper (CBS) flows. The deployment opportunities increase when multiple mechanisms are available, as flows with less stringent requirements can utilize mechanisms offering lower guarantees. In the context of using flexibility metrics, it is imperative to include competing mechanisms in the flexibility quantification, to ensure multi-mechanism usage is considered.

7.1.2 *Conclusions*

Time-Sensitive Networking is an enabling technology for achieving deterministic communication in standard Ethernet. As the goals of the Industrial Internet of Things (IIoT) and Industry 4.0 demand both flexibility and determinism in data communication, there is a need to enhance adaptability when deploying TSN. However, the inherent flexibility of Ethernet diminishes when real-time guarantees are provided through the deployment of TSN mechanisms. Thus, the adaptability of real-time data communication at runtime becomes a critical issue.

In this thesis, we introduce a novel notion of flexibility, the flexcurve, designed to enhance and quantify the flexibility of the TSN scheduling process. This addresses a critical gap in achieving both deterministic communication and online adaptability.

We evaluate the flexcurve measure by analyzing its runtime behavior using various formulations. Empirical results confirm that these match the expected runtime behavior. Notably, through flexcurve aggregations, we achieve constant lookup times when schedules exhibit similar fragmentations.

To integrate the flexcurve into the scheduling process, we introduce a search algorithm to identify flow eligibility candidates. We assess the algorithm's runtime on two distinct topologies: a worst-case line topology and a hierarchically complex machine topology. Our findings demonstrate sub-second runtimes for the machine topology, which are significantly improved compared to traditional scheduling and incremental SMT-based approaches. To accommodate additional parameters in the flexcurve, our evaluations indicate increased runtime requirements for computing a deadline-aware flexcurve, relative to the limited formulation. However, this runtime decreases as the schedule capacity increases. The approximation to select flexcurve optimal candidates, shows up to 96x in runtime improvements, compared to naive recomputations.

We extend the flexcurve to accommodate requirements beyond scheduled traffic. Given the diversity of the TSN mechanism ecosystem, mechanisms can be selectively chosen to meet application requirements. This selective choice avoids the necessity of selecting mechanisms that provide more stringent guarantees than necessary, thereby preventing the reservation of excessive resources. The extension of the flexcurve to include additional traffic types, such as CBS, reflects the interplay between mechanisms and their impact on scheduled traffic flexibility. Further, by extending deployment support to enable deployment on other forwarding hardware not directly supporting TSN, such as programmable forwarding hardware and programmable schedulers, we further enhance the mechanism ecosystem. This enhancement increases the versatility of network configurations and adaptability across various networking environments.

This thesis shows and establishes a comprehensive framework for quantifying flexibility in TSN networks. This framework can be extended to include multiple scheduling mechanisms, and reflect the requirements that are specified by flows of interest. This adaptability ensures that the metric reflects the true potential for accommodating varying network demands and hardware capabilities.

7.2 OUTLOOK

In this section, we identify potential avenues for future research and outline possible approaches to address these open problems.

The flexcurve metric incorporates various flow parameters, depending on user specifications. We have provided specific formulations of the metric to include frame sizes and deadlines. Further extension of these formulations to encompass additional parameters, such as specific flow periods or multiple destinations, would allow for a more precise representation of application flow requirements. Incorporating arbitrary flow periods addresses a significant limitation of the flexcurve formulations, as discussed in this thesis, and would facilitate a comprehensive view of future scheduled traffic. A potential approach to include arbitrary flow periods could involve extending Algorithm 2 to identify eligible candidates. The algorithm can be modified to accommodate arbitrary cycles, as the current version assumes cycles equal to the hyperperiod. One possible method could be to enable the shifting of multiple frame instances that are separated by a fixed cycle period of slots. The inclusion of multi-destination capabilities, implying multiple listeners per flow, necessitates multicast support. Currently, the flexcurve does not accommodate multicast, a limitation that would necessitate substantial modifications to its definition. Moreover, the potential for multiple bottlenecks along diverging paths raises further questions. It is unclear whether a multicast-aware flexcurve should resemble a unicast flexcurve by reflecting shared bottlenecks, or whether it would be more effective to segregate paths directed toward individual listeners.

We have extended the flexcurve to reflect the concurrent deployment of the TSN CBS mechanism. Extending the flexcurve to include the remaining mechanisms, such as Asynchronous Traffic Shaping (ATS), could facilitate more possibilities for the simultaneous deployment of TSN mechanisms when the flexcurve is used to quantify scheduled traffic flexibility. The method could be akin to integrating CBS; it should allow the limiting of eligibility candidates based on the criteria of other traffic type flows. Naturally, the configuration becomes increasingly complex with the concurrent deployment of multiple mechanisms. Additionally, it is imperative to ensure that the mechanisms adequately account for their interferences, provided that these have not already been considered.

The flexcurve metric quantifies the flexibility of scheduled data frame sequences. Further research is needed to examine the utility of this metric in other domains apart from TSN. It is essential to determine both the feasibility of adapting the flexcurve metric and the effectiveness of its application for other domains. Areas such as manu-

facturing systems and public transportation, where scheduled sequences are common, could particularly benefit from this exploration.

Lastly, in TSN, the implementation of Frame Replication and Elimination for Reliability (FRER) mechanisms facilitates the introduction of data redundancy over multiple paths to enable instantaneous failover. It is essential to explore how FRER can be incorporated into the flexcurve framework when applications necessitate physical redundancy. There is a potential opportunity to combine the use of FRER with multicast support, as both require modifications to the formulations of the flexcurve. While multicast addresses multiple listeners via one diverging path, FRER aims to establish disjoint paths to one or more destinations. Both, FRER and multicast, necessitate the consideration of additional routing specifications.

ACKNOWLEDGMENTS

Work described in this thesis was funded by the German Research Foundation (DFG) within the Collaborative Research Centre 1053 "MAKI". Additionally, work in this thesis was aided by Robert Bosch GmbH, within the T3 subproject of MAKI.

BIBLIOGRAPHY

[1] Industry IoT Consortium (IIC). *Time Sensitive Networks for Flexible Manufacturing Testbed Characterization and Mapping of Converged Traffic Types*. Tech. rep. Mar. 2019. URL: https://www.iiconsortium.org/pdf/IIC_TSN_Testbed_Char_Mapping_of_Converged_Traffic_Types_Whitepaper_20180328.pdf (Last accessed on Jan. 30, 2024).

[2] Anurag Agrawal and Changhoon Kim. "Intel Tofino2 – A 12.9Tbps P4-Programmable Ethernet Switch." In: *2020 IEEE Hot Chips 32 Symposium (HCS)*. IEEE, 2020, pp. 1–32. DOI: 10.1109/HCS49909.2020.9220636.

[3] Albert Gran Alcoz, Alexander Dietmüller, and Laurent Vanbever. "SP-PIFO: Approximating Push-In-First-Out Behaviors using Strict-Priority Queues." In: *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*. NSDI'20. USENIX Association, 2020, pp. 59–76.

[4] Bastian Alt, Markus Weckesser, Christian Becker, Matthias Hollick, Sounak Kar, Anja Klein, Robin Klose, Roland Kluge, Heinz Koeppl, Boris Koldehofe, Wasiur R. Khudabukhsh, Manisha Luthra, Mahdi Mousavi, Max Mühlhäuser, Martin Pfannemüller, Amr Rizk, Andy Schürr, and Ralf Steinmetz. "Transitions: A Protocol-Independent View of the Future Internet." In: *Proceedings of the IEEE* 107.4 (2019), pp. 835–846. DOI: 10.1109/JPROC.2019.2895964.

[5] Onur Altintas, Y. Atsumi, and T. Yoshida. "Urgency-based round robin: a new scheduling discipline for packet switching networks." In: *ICC '98. 1998 IEEE International Conference on Communications. Conference Record. Affiliated with SUPERCOMM'98 (Cat. No.98CH36220)*. Vol. 2. 1998, 1179–1184 vol.2. DOI: 10.1109/ICC.1998.685195.

[6] Anna Arestova, Wojciech Baron, Kai-Steffen J. Hielscher, and Reinhard German. "ITANS: Incremental Task and Network Scheduling for Time-Sensitive Networks." In: *IEEE Open Journal of Intelligent Transportation Systems* 3 (2022), pp. 369–387. DOI: 10.1109/OJITS.2022.3171072.

[7] Péter Babarczi, Markus Klügel, Alberto Martínez Alba, Mu He, Johannes Zerwas, Patrick Kalmbach, Andreas Blenk, and Wolfgang Kellerer. "A mathematical framework for measuring network flexibility." In: *Journal of Computer Communications* 164 (2020), pp. 13–24. DOI: 10.1016/j.comcom.2020.09.014.

[8] Thomas Bach, Muhammad Adnan Tariq, Boris Koldehofe, and Kurt Rothermel. "A cost efficient scheduling strategy to guarantee probabilistic workflow deadlines." In: *2015 International Conference and Workshops on Networked Systems (NetSys)*. IEEE, 2015, pp. 1–8. DOI: 10.1109/NetSys.2015.7089072.

[9] Mohammadreza Barzegaran and Paul Pop. “Communication Scheduling for Control Performance in TSN-Based Fog Computing Platforms.” In: *Journal of IEEE Access* 9 (2021), pp. 50782–50797. DOI: 10.1109/ACCESS.2021.3069142.

[10] Rudy Belliardi, Josef Dorr, Thomas Enzinger, Florian Essler, János Farkas, Mark Hantel, Maximilian Riegel, Marius-Petru Stanica, Guenter Steindl, Reiner Wamßer, Karl Weber, and Steven A. Zuponcic. *Use Cases IEC/IEEE 60802 v1.3*. Sept. 2018. URL: http://www.ieee802.org/1/files/public/docs2018/60802-industrial-use-cases-0918-v13.pdf (Last accessed on Jan. 23, 2024).

[11] Lucia Lo Bello. “Novel trends in automotive networks: A perspective on Ethernet and the IEEE Audio Video Bridging.” In: *Proceedings of the 2014 IEEE Emerging Technology and Factory Automation (ETFA)*. 2014, pp. 1–8. DOI: 10.1109/ETFA.2014.7005251.

[12] Aldin Berisa, Luxi Zhao, Silviu S. Craciunas, Mohammad Ashjaei, Saad Mubeen, Masoud Daneshtalab, and Mikael Sjödin. “AVB-aware Routing and Scheduling for Critical Traffic in Time-sensitive Networks with Preemption.” In: *Proceedings of the 30th International Conference on Real-Time Networks and Systems*. RTNS ’22. Association for Computing Machinery, 2022, pp. 207–218. DOI: 10.1145/3534879.3534926.

[13] Randeep Bhatia, T.V. Lakshman, Mustafa F. Ozkoc, and Shivendra Panwar. “FlowToss: Fast Wait-Free Scheduling of Deterministic Flows in Time Synchronized Networks.” In: *2021 IFIP Networking Conference (IFIP Networking)*. IEEE, 2021, pp. 1–6. DOI: 10.23919/IFIPNetworking52078.2021.9472838.

[14] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, and David Walker. “P4: Programming Protocol-Independent Packet Processors.” In: *Journal of SIGCOMM Computer Communication Review* 44.3 (2014), pp. 87–95. DOI: 10.1145/2656877.2656890.

[15] Bochra Boughzala, Christoph Gärtner, and Boris Koldehofe. “Window-Based Parallel Operator Execution with in-Network Computing.” In: *Proceedings of the 16th ACM International Conference on Distributed and Event-Based Systems*. DEBS ’22. Association for Computing Machinery, 2022, pp. 91–96. DOI: 10.1145/3524860.3539804.

[16] Dietmar Bruckner, Marius-Petru Stănică, Richard Blair, Sebastian Schriegel, Stephan Kehrer, Maik Seewald, and Thilo Sauter. “An Introduction to OPC UA TSN for Industrial Communication Systems.” In: *Proceedings of the IEEE* 107.6 (2019), pp. 1121–1131. DOI: 10.1109/JPROC.2018.2888703.

[17] Daniel Bujosa, Mohammad Ashjaei, Alessandro V. Papadopoulos, Thomas Nolte, and Julian Proenza. “HERMES: Heuristic Multi-queue Scheduler for TSN Time-Triggered Traffic with Zero Reception Jitter Capabilities.” In: *Proceedings of the 30th International Conference on Real-Time Networks and Systems*. Association for Computing Machinery, 2022, pp. 70–80. DOI: 10.1145/3534879.3534906.

[18] 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 the 24th International Conference on Real-Time Networks and Systems*. RTNS '16. Association for Computing Machinery, 2016, pp. 183–192. DOI: 10.1145/2997465.2997470.

[19] Rene L. Cruz. "A calculus for network delay. I. Network elements in isolation." In: *IEEE Transactions on Information Theory* 37.1 (1991), pp. 114–131. DOI: 10.1109/18.61109.

[20] Joan Adrià Ruiz De Azua and Marc Boyer. "Complete modelling of AVB in Network Calculus Framework." In: *Proceedings of the 22nd International Conference on Real-Time Networks and Systems*. RTNS '14. Association for Computing Machinery, 2014, pp. 55–64. DOI: 10.1145/2659787.2659810.

[21] 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." In: *Journal of ACM Computing Surveys* 55.2 (2022). DOI: 10.1145/3487330.

[22] Frank Dürr and Naresh Ganesh Nayak. "No-Wait Packet Scheduling for IEEE Time-Sensitive Networks (TSN)." In: *Proceedings of the 24th International Conference on Real-Time Networks and Systems*. RTNS '16. Association for Computing Machinery, 2016, pp. 203–212. DOI: 10.1145/2997465.2997494.

[23] Jonathan Falk, David Hellmanns, Ben Carabelli, Naresh Nayak, Frank Dürr, Stephan Kehrer, and Kurt Rothermel. "NeSTiNg: Simulating IEEE time-sensitive networking (TSN) in OMNeT++." In: *2019 International Conference on Networked Systems (NetSys)*. IEEE, 2019, pp. 1–8. DOI: 10.1109/NetSys.2019.8854500.

[24] Markus Fidler and Amr Rizk. "A Guide to the Stochastic Network Calculus." In: *IEEE Communications Surveys & Tutorials* 17.1 (2015), pp. 92–105. DOI: 10.1109/COMST.2014.2337060.

[25] Wen Gao, Borui Zhao, and Xu Mao. "Research on Incremental Scheduling Backtracking Algorithm for Time-triggered Ethernet." In: *2020 2nd International Conference on Advances in Computer Technology, Information Science and Communications (CTISC)*. IEEE, 2020, pp. 75–79. DOI: 10.1109/CTISC49998.2020.00019.

[26] Christoph Gärtner, Amr Rizk, Boris Koldehofe, René Guillaume, Ralf Kundel, and Ralf Steinmetz. "On the Incremental Reconfiguration of Time-sensitive Networks at Runtime." In: *2022 IFIP Networking Conference (IFIP Networking)*. IEEE, 2022, pp. 1–9. DOI: 10.23919/IFIPNetworking55013.2022.9829815.

[27] Christoph Gärtner, Amr Rizk, Boris Koldehofe, René Guillaume, Ralf Kundel, and Ralf Steinmetz. "Demo: Flexibility-Aware Network Management of Time-Sensitive Flows." In: *Proceedings of the ACM SIGCOMM 2023 Conference*. ACM SIGCOMM '23. Association for Computing Machinery, 2023, pp. 1176–1178. DOI: 10.1145/3603269.3610869.

[28] Christoph Gärtner, Amr Rizk, Boris Koldehofe, René Guillaume, Ralf Kundel, and Ralf Steinmetz. "Fast incremental reconfiguration of dynamic time-sensitive networks at runtime." In: *Journal of Computer Networks* 224 (2023), p. 109606. DOI: https://doi.org/10.1016/j.comnet.2023.109606.

[29] Christoph Gärtner, Amr Rizk, Boris Koldehofe, Rhaban Hark, René Guillaume, Ralf Kundel, and Ralf Steinmetz. "POSTER: Leveraging PIFO Queues for Scheduling in Time-Sensitive Networks." In: *2021 IEEE International Symposium on Local and Metropolitan Area Networks (LANMAN)*. IEEE, 2021, pp. 1–2. DOI: 10.1109/LANMAN52105.2021.9478796.

[30] Christoph Gärtner, Amr Rizk, Boris Koldehofe, Rhaban Hark, René Guillaume, and Ralf Steinmetz. "Leveraging Flexibility of Time-Sensitive Networks for dynamic Reconfigurability." In: *2021 IFIP Networking Conference (IFIP Networking)*. IEEE, 2021, pp. 1–6. DOI: 10.23919/IFIPNetworking52078.2021.9472834.

[31] Voica Gavriluţ, Luxi Zhao, Michael L. Raagaard, and Paul Pop. "AVB-Aware Routing and Scheduling of Time-Triggered Traffic for TSN." In: *IEEE Access* 6 (2018), pp. 75229–75243. DOI: 10.1109/ACCESS.2018.2883644.

[32] Pegah Golchin, Chengbo Zhou, Pratyush Agnihotri, Mehrdad Hajizadeh, Ralf Kundel, and Ralf Steinmetz. "CML-IDS: Enhancing Intrusion Detection in SDN Through Collaborative Machine Learning." In: *2023 19th International Conference on Network and Service Management (CNSM)*. IEEE. 2023, pp. 1–9.

[33] Alexej Grigorjew, Nicholas Gray, and Tobias Hoßfeld. "Dynamic Real-Time Stream Reservation with TAS and Shared Time Windows." In: *2021 IFIP Networking Conference (IFIP Networking)*. IEEE, 2021, pp. 1–6. DOI: 10.23919/IFIPNetworking52078.2021.9472800.

[34] Alexej Grigorjew, Florian Metzger, Tobias Hoßfeld, Johannes Specht, Franz-Josef Götz, Feng Chen, and Jürgen Schmitt. "Constant Delay Switching: Asynchronous Traffic Shaping with Jitter Control." In: *2022 IFIP Networking Conference (IFIP Networking)*. IEEE, June 2022, pp. 1–9. DOI: 10.23919/IFIPNetworking55013.2022.9829777.

[35] Juha Heinanen and Roch Guerin. *A Single Rate Three Color Marker*. RFC 2697. Sept. 1999. DOI: 10.17487/RFC2697. URL: https://www.rfc-editor.org/info/rfc2697 (Last accessed on Apr. 2, 2024).

[36] Jia Huang, Jan Olaf Blech, Andreas Raabe, Christian Buckl, and Alois Knoll. "Static scheduling of a Time-Triggered Network-on-Chip based on SMT solving." In: *2012 Design, Automation Test in Europe Conference Exhibition (DATE)*. IEEE, 2012, pp. 509–514. DOI: 10.1109/DATE.2012.6176522.

[37] IBM. *What is Industry 4.0?* URL: https://www.ibm.com/topics/industry-4-0 (Last accessed on Jan. 23, 2024).

[38] "IEEE Draft Standard for Local and Metropolitan Area Networks–Bridges and Bridged Networks Amendment 38: Configuration Enhancements for Time-Sensitive Networking." In: *IEEE P802.1Qdj/D2.0, November 2023* (2023), pp. 1–49.

[39] "IEEE Standard for a Precision Clock Synchronization Protocol for Networked Measurement and Control Systems." In: *IEEE Std 1588-2008 (Revision of IEEE Std 1588-2002)* (2008), pp. 1–269. DOI: 10.1109/IEEESTD.2008.4579760.

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

[41] "IEEE Standard for Local and metropolitan area networks – Bridges and Bridged Networks – Amendment 26: Frame Preemption." In: *IEEE Std 802.1Qbu-2016 (Amendment to IEEE Std 802.1Q-2014)* (2016), pp. 1–52. DOI: 10.1109/IEEESTD.2016.7553415.

[42] "IEEE Standard for Local and Metropolitan Area Networks–Bridges and Bridged Networks." In: *IEEE Std 802.1Q-2022 (Revision of IEEE Std 802.1Q-2018)* (2022), pp. 1–2163. DOI: 10.1109/IEEESTD.2022.10004498.

[43] "IEEE Standard for Local and Metropolitan Area Networks–Bridges and Bridged Networks - Amendment 34: Asynchronous Traffic Shaping." In: *IEEE Std 802.1Qcr-2020 (Amendment to IEEE Std 802.1Q-2018 as amended by IEEE Std 802.1Qcp-2018, IEEE Std 802.1Qcc-2018, IEEE Std 802.1Qcy-2019, and IEEE Std 802.1Qcx-2020)* (2020), pp. 1–151. DOI: 10.1109/IEEESTD.2020.9253013.

[44] "IEEE Standard for Local and Metropolitan Area Networks–Bridges and Bridged Networks – Amendment 31: Stream Reservation Protocol (SRP) Enhancements and Performance Improvements." In: *IEEE Std 802.1Qcc-2018 (Amendment to IEEE Std 802.1Q-2018 as amended by IEEE Std 802.1Qcp-2018)* (2018), pp. 1–208. DOI: 10.1109/IEEESTD.2018.8514112.

[45] "IEEE Standard for Local and metropolitan area networks–Bridges and Bridged Networks–Amendment 28: Per-Stream Filtering and Policing." In: *IEEE Std 802.1Qci-2017 (Amendment to IEEE Std 802.1Q-2014 as amended by IEEE Std 802.1Qca-2015, IEEE Std 802.1Qcd-2015, IEEE Std 802.1Q-2014/Cor 1-2015, IEEE Std 802.1Qbv-2015, IEEE Std 802.1Qbu-2016, and IEEE Std 802.1Qbz-2016)* (2017), pp. 1–65. DOI: 10.1109/IEEESTD.2017.8064221.

[46] "IEEE Standard for Local and metropolitan area networks–Frame Replication and Elimination for Reliability." In: *IEEE Std 802.1CB-2017* (2017), pp. 1–102. DOI: 10.1109/IEEESTD.2017.8091139.

[47] "IEEE Standard for Local and Metropolitan Area Networks–Timing and Synchronization for Time-Sensitive Applications." In: *IEEE Std 802.1AS-2020 (Revision of IEEE Std 802.1AS-2011)* (2020), pp. 1–421. DOI: 10.1109/IEEESTD.2020.9121845.

[48] *Intel® Tofino™* 2. URL: https://www.intel.com/content/www/us/en/products/details/network-io/intelligent-fabric-processors/tofino-2.html (Last accessed on Apr. 23, 2024).

[49] Enio Kaljic, Almir Maric, Pamela Njemcevic, and Mesud Hadzialic. "A Survey on Data Plane Flexibility and Programmability in Software-Defined Networking." In: *IEEE Access* 7 (2019), pp. 47804–47840. DOI: 10.1109/ACCESS.2019.2910140.

[50] Manolis Katevenis, Stefanos Sidiropoulos, and Costas Courcoubetis. "Weighted round-robin cell multiplexing in a general-purpose ATM switch chip." In: *IEEE Journal on Selected Areas in Communications* 9.8 (1991), pp. 1265–1279. DOI: 10.1109/49.105173.

[51] Naga Katta, Mukesh Hira, Changhoon Kim, Anirudh Sivaraman, and Jennifer Rexford. "HULA: Scalable Load Balancing Using Programmable Data Planes." In: *Proceedings of the Symposium on SDN Research*. SOSR '16. Association for Computing Machinery, 2016. DOI: 10.1145/2890955.2890968.

[52] Diego Kreutz, Fernando M. V. Ramos, Paulo Esteves Veríssimo, Christian Esteve Rothenberg, Siamak Azodolmolky, and Steve Uhlig. "Software-Defined Networking: A Comprehensive Survey." In: *Proceedings of the IEEE* 103.1 (2015), pp. 14–76. DOI: 10.1109/JPROC.2014.2371999.

[53] VC Kumar. *The state of functional safety in Industry 4.0*. Tech. rep. Texas Instruments, Dec. 2018. URL: https://www.ti.com/lit/fs/spry329/spry329.pdf (Last accessed on Jan. 23, 2024).

[54] Ralf Kundel, Jeremias Blendin, Tobias Viernickel, Boris Koldehofe, and Ralf Steinmetz. "P4-CoDel: Active Queue Management in Programmable Data Planes." In: *2018 IEEE Conference on Network Function Virtualization and Software Defined Networks (NFV-SDN)*. 2018, pp. 1–4. DOI: 10.1109/NFV-SDN.2018.8725736.

[55] Ralf Kundel, Christoph Gärtner, Manisha Luthra, Sukanya Bhowmik, and Boris Koldehofe. "Flexible Content-based Publish/Subscribe over Programmable Data Planes." In: *NOMS 2020 - 2020 IEEE/IFIP Network Operations and Management Symposium*. 2020, pp. 1–5. DOI: 10.1109/NOMS47738.2020.9110381.

[56] Ralf Kundel, Nehal Baganal Krishna, Christoph Gärtner, Tobias Meuser, and Amr Rizk. "Poster: Reverse-Path Congestion Notification: Accelerating the Congestion Control Feedback Loop." In: *2021 IEEE 29th International Conference on Network Protocols (ICNP)*. IEEE, 2021, pp. 1–2. DOI: 10.1109/ICNP52444.2021.9651961.

[57] Ralf Kundel, Leonhard Nobach, Hans-Joerg Kolbe, Tobias Meuser, and Ralf Steinmetz. "FPGA-assisted Massive Packet Queueing and Traffic Shaping at the Network Edge." In: *2022 IEEE 30th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. 2022, p. 1. DOI: 10.1109/FCCM53951.2022.9786068.

[58] Ralf Kundel, Amr Rizk, Jeremias Blendin, Boris Koldehofe, Rhaban Hark, and Ralf Steinmetz. "P4-CoDel: Experiences on Programmable Data Plane Hardware." In: *ICC 2021 - IEEE International Conference on Communications*. 2021, pp. 1–6. DOI: 10.1109/ICC42927.2021.9500943.

[59] Thomas R. Kurfess, Christopher Saldana, Kyle Saleeby, and Mahmoud Parto Dezfouli. "A Review of Modern Communication Technologies for Digital Manufacturing Processes in Industry 4.0." In: *Journal of Manufacturing Science and Engineering* 142.11 (Sept. 2020), p. 110815. DOI: 10.1115/1.4048206.

[60] Jean-Yves Le Boudec and Patrick Thiran. *Network Calculus: A Theory of Deterministic Queuing Systems for the Internet*. Springer Berlin Heidelberg, 2001. ISBN: 978-3-540-42184-9. DOI: 10.1007/3-540-45318-0.

[61] Dong-Jun Lee. "Incremental Routing and Scheduling Using Multipath and Nonzero Jitter Bound for IEEE 802.1 Qbv Time Aware Shaper." In: *Journal of IEEE Access* 11 (2023), pp. 25035–25049. DOI: 10.1109/ACCESS.2023.3255416.

[62] Joseph Y.-T. Leung. "A new algorithm for scheduling periodic, real-time tasks." In: *Algorithmica* 4.1 (June 1989), pp. 209–219. DOI: 10.1007/BF01553887.

[63] C. L. Liu and James W. Layland. "Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment." In: *Journal of the ACM* 20.1 (1973), pp. 46–61. DOI: 10.1145/321738.321743.

[64] Xiaoyu Liu, Chi Xu, and Haibin Yu. "Network Calculus-based Modeling of Time Sensitive Networking Shapers for Industrial Automation Networks." In: *2019 11th International Conference on Wireless Communications and Signal Processing (WCSP)*. 2019, pp. 1–7. DOI: 10.1109/WCSP.2019.8927901.

[65] Lucia Lo Bello and Wilfried Steiner. "A Perspective on IEEE Time-Sensitive Networking for Industrial Communication and Automation Systems." In: *Proceedings of the IEEE* 107.6 (2019), pp. 1094–1120. DOI: 10.1109/JPROC.2019.2905334.

[66] David Zhe Lou, Jan Holler, Cliff Whitehead, Sari Germanos, Michael Hilgner, and Wei Qiu. *Industrial Networking Enabling IIoT Communication*. Tech. rep. Industry IoT Consortium (IIC), Aug. 2018. URL: https://www.iiconsortium.org/pdf/Industrial_Networking_Enabling_IIoT_Communication_2018_08_29.pdf (Last accessed on Jan. 30, 2024).

[67] Lisa Maile, Kai-Steffen J. Hielscher, and Reinhard German. "Delay-Guaranteeing Admission Control for Time-Sensitive Networking Using the Credit-Based Shaper." In: *IEEE Open Journal of the Communications Society* 3 (2022), pp. 1834–1852. DOI: 10.1109/OJCOMS.2022.3212939.

[68] Dorin Maxim and Ye-Qiong Song. "Delay analysis of AVB traffic in time-sensitive networks (TSN)." In: *Proceedings of the 25th International Conference on Real-Time Networks and Systems*. RTNS '17. Association for Computing Machinery, 2017, pp. 18–27. DOI: 10.1145/3139258.3139283.

[69] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. "OpenFlow: enabling innovation in campus networks." In: *Journal of ACM SIGCOMM Computer Communication Review* 38.2 (2008), pp. 69–74. DOI: 10.1145/1355734.1355746.

[70] Ehsan Mohammadpour, Eleni Stai, Maaz Mohiuddin, and Jean-Yves Le Boudec. "Latency and Backlog Bounds in Time-Sensitive Networking with Credit Based Shapers and Asynchronous Traffic Shaping." In: *2018 30th International Teletraffic Congress (ITC 30)*. IEEE, Sept. 2018. DOI: 10.1109/itc30.2018.10053.

[71] Naresh Ganesh Nayak, Frank Dürr, and Kurt Rothermel. "Incremental flow scheduling and routing in time-sensitive software-defined networks." In: *Journal of IEEE Transactions on Industrial Informatics* 14.5 (2017), pp. 2066–2075. DOI: 10.1109/tii.2017.2782235.

[72] Naresh Ganesh Nayak, Frank Dürr, and Kurt Rothermel. "Routing algorithms for IEEE802.1Qbv networks." In: *Journal of SIGBED Review* 15.3 (2018), pp. 13–18. DOI: 10.1145/3267419.3267421.

[73] Don Pannell. *AVB Latency Math*. Nov. 2010. URL: https://www.ieee802.org/1/files/public/docs2010/BA-pannell-latency-math-1110-v5.pdf (Last accessed on Mar. 30, 2024).

[74] Michael Pinedo. *Scheduling: Theory, Algorithms, and Systems*. Fifth Edition. Springer Cham, Feb. 2016. ISBN: 978-3-319-26580-3. DOI: 10.1007/978-3-319-26580-3.

[75] Paul Pop, Michael Lander Raagaard, Marina Gutierrez, and Wilfried Steiner. "Enabling Fog Computing for Industrial Automation Through Time-Sensitive Networking (TSN)." In: *IEEE Communications Standards Magazine* 2.2 (2018), pp. 55–61. DOI: 10.1109/MCOMSTD.2018.1700057.

[76] *PROFINET - The Leading Industrial Ethernet Protocol*. URL: https://www.profinet.com (Last accessed on Apr. 8, 2024).

[77] *Profinet und TSN sind Enabler für die Industrie 4.0*. July 2021. URL: https://www.sps-magazin.de/protokolle-standards/profinet-und-tsn-sind-enabler-fuer-die-industrie-4-0/ (Last accessed on Jan. 23, 2024).

[78] Gunnar Prytz. "A performance analysis of EtherCAT and PROFINET IRT." In: *2008 IEEE International Conference on Emerging Technologies and Factory Automation*. 2008, pp. 408–415. DOI: 10.1109/ETFA.2008.4638425.

[79] Rene Queck. "Analysis of Ethernet AVB for automotive networks using Network Calculus." In: *2012 IEEE International Conference on Vehicular Electronics and Safety (ICVES 2012)*. 2012, pp. 61–67. DOI: 10.1109/ICVES.2012.6294261.

[80] Michael Lander Raagaard and Paul Pop. *Optimization Algorithms for the Scheduling of IEEE 802.1 Time-Sensitive Networking (TSN)*. Tech. rep. DTU Compute, Technical University of Denmark, Jan. 2017. URL: http://www2.compute.dtu.dk/~paupo/publications/Raagaard2017aa-Optimization%20algorithms%20for%20th-.pdf (Last accessed on Apr. 5, 2024).

[81] Ram Rachamadugu, Udayan Nandkeolyar, and Tom Schriber. "Scheduling with Sequencing Flexibility*." In: *Journal of Decision Sciences* 24.2 (1993), pp. 315–342. DOI: 10.1111/j.1540-5915.1993.tb00477.x.

[82] Sriram Ramabhadran and Joseph Pasquale. "Stratified round Robin: a low complexity packet scheduler with bandwidth fairness and bounded delay." In: *Proceedings of the 2003 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*. SIGCOMM '03. Association for Computing Machinery, 2003, pp. 239–250. DOI: 10.1145/863955.863983.

[83] Aellison Cassimiro T. dos Santos, Ben Schneider, and Vivek Nigam. "TSNSCHED: Automated Schedule Generation for Time Sensitive Networking." In: *2019 Formal Methods in Computer Aided Design (FMCAD)*. IEEE, 2019, pp. 69–77. DOI: 10.23919/FMCAD.2019.8894249.

[84] Helen Schonenberg, Ronny Mans, Nick Russell, Nataliya Mulyar, and Wil van der Aalst. "Process Flexibility: A Survey of Contemporary Approaches." In: *Advances in Enterprise Engineering I*. Springer, 2008, pp. 16–30. DOI: 10.1007/978-3-540-68644-6_2.

[85] Sebastian Schriegel and Jürgen Jasperneite. "A Migration Strategy for Profinet Toward Ethernet TSN-Based Field-Level Communication: An Approach to Accelerate the Adoption of Converged IT/OT Communication." In: *IEEE Industrial Electronics Magazine* 15.4 (2021), pp. 43–53. DOI: 10.1109/MIE.2020.3048925.

[86] Andrea Krasa Sethi and Suresh Pal Sethi. "Flexibility in manufacturing: A survey." In: *International Journal of Flexible Manufacturing Systems* 2.4 (1990), pp. 289–328. DOI: 10.1007/BF00186471.

[87] Eva Shayo, Prosper Mafole, and Alfred Mwambela. "A survey on time division multiple access scheduling algorithms for industrial networks." In: *Journal of SN Applied Sciences* 2.12 (Dec. 2020), p. 2140. DOI: 10.1007/s42452-020-03923-4.

[88] M. Shreedhar and G. Varghese. "Efficient fair queuing using deficit round-robin." In: *IEEE/ACM Transactions on Networking* 4.3 (1996), pp. 375–385. DOI: 10.1109/90.502236.

[89] Anirudh Sivaraman, Suvinay Subramanian, Mohammad Alizadeh, Sharad Chole, Shang-Tse Chuang, Anurag Agrawal, Hari Balakrishnan, Tom Edsall, Sachin Katti, and Nick McKeown. "Programmable Packet Scheduling at Line Rate." In: *Proceedings of the 2016 ACM SIGCOMM Conference*. SIGCOMM '16. Association for Computing Machinery, 2016, pp. 44–57. DOI: 10.1145/2934872.2934899.

[90] Wen Song, Xinyang Chen, Qiqiang Li, and Zhiguang Cao. "Flexible Job-Shop Scheduling via Graph Neural Network and Deep Reinforcement Learning." In: *IEEE Transactions on Industrial Informatics* 19.2 (2023), pp. 1600–1610. DOI: 10.1109/TII.2022.3189725.

[91] Johannes Specht and Soheil Samii. "Urgency-Based Scheduler for Time-Sensitive Switched Ethernet Networks." In: *2016 28th Euromicro Conference on Real-Time Systems (ECRTS)*. 2016, pp. 75–85. DOI: 10.1109/ECRTS.2016.27.

[92] Andrew S. Tanenbaum and David J. Wetherall. *Computer Networks*. 5th ed. Pearson Education, Inc., publishing as Prentice Hall, 2011. ISBN: 978-0-13-212695-3.

[93] *Time-Triggered Ethernet*. URL: https://www.tttech.com/explore/time-triggered-ethernet (Last accessed on Apr. 8, 2024).

[94] Christian Wernecke, Helge Parzyjegla, Gero Mühl, Peter Danielis, and Dirk Timmermann. "Realizing Content-Based Publish/Subscribe with P4." In: *2018 IEEE Conference on Network Function Virtualization and Software Defined Networks (NFV-SDN)*. 2018, pp. 1–7. DOI: 10.1109/NFV-SDN.2018.8725641.

[95] Bundesministerium für Wirtschaft und Klimaschutz (BMWK). *Fortschrittsbericht 2023 Industrie 4.0: Auf dem Weg zur intelligent vernetzten Industrie*. May 2023. URL: https://www.plattform-i40.de/IP/Redaktion/DE/Downloads/Publikation/2023-fortschrittsbericht.html (Last accessed on Jan. 23, 2024).

[96] Martin Wollschlaeger, Thilo Sauter, and Juergen Jasperneite. "The Future of Industrial Communication: Automation Networks in the Era of the Internet of Things and Industry 4.0." In: *Journal of IEEE Industrial Electronics Magazine* 11.1 (Mar. 2017). Conference Name: IEEE Industrial Electronics Magazine, pp. 17–27. DOI: 10.1109/MIE.2017.2649104.

[97] *Z3*. URL: https://github.com/Z3Prover/z3 (Last accessed on Apr. 8, 2024).

[98] Luxi Zhao, Paul Pop, and Sebastian Steinhorst. "Quantitative Performance Comparison of Various Traffic Shapers in Time-Sensitive Networking." In: *IEEE Transactions on Network and Service Management* 19.3 (2022), pp. 2899–2928. DOI: 10.1109/tnsm.2022.3180160.

[99] Luxi Zhao, Paul Pop, Zhong Zheng, Hugo Daigmorte, and Marc Boyer. "Latency Analysis of Multiple Classes of AVB Traffic in TSN With Standard Credit Behavior Using Network Calculus." In: *IEEE Transactions on Industrial Electronics* 68.10 (2021), pp. 10291–10302. DOI: 10.1109/TIE.2020.3021638.

[100] Chengbo Zhou, Christoph Gärtner, Amr Rizk, Boris Koldehofe, Björn Scheuermann, and Ralf Kundel. "RDA: Residence Delay Aggregation for Time-Sensitive Networking." In: *Proceedings of 2024 IEEE Network Operations and Management Symposium (NOMS 2024)*. 2024. DOI: 10.1109/NOMS59830.2024.10574998.

All web pages cited in this work have been checked on the date given in the reference.

A APPENDIX

A.1 FLEXIBILITY-AWARE CONTROLLER IMPLEMENTATION

As part of our demonstration in [27], we implemented a combined Centralized User Configuration (CUC)/Centralized Network Configuration (CNC) Time-Sensitive Networking (TSN) controller. This controller, implemented in Python, integrates the flexibility mechanisms described in this thesis. This implementation embodies the concepts presented in Chapter 3. Below, we detail the specific features of this prototype controller.

Application API

The controller is accessible to users or deployed applications via websockets. This choice facilitates the creation of straightforward Web-demo interfaces, which can be programmed using HTML/CSS and JavaScript to interact directly with the Python backend.

The Websocket Application Programming Interface (API) manages asynchronous flow requests. The controller issues a **`scheduleUpdate`** event in response, which, when linked with physical hardware, incorporates the TSN configuration's **`configChangeTime`** to facilitate the initiation of new flows.

Controller

The controller architecture is divided into several components: User API, network topology, flow management, schedule management with schedulers and flexibility enhancements, and hardware abstraction or drivers. Designed as a prototype, this controller showcases flexibility features without adhering to the standard interface or segregation between the CUC and CNC as outlined in the TSN standard.

- The **User API** component handles requests from applications and interfaces with other controller components.
- The **Network topology** component manages the network layout, provides mechanisms to identify feasible and shortest routes between nodes, and links nodes to physical devices for configuration deployment.
- The **Flow management** component maintains records of both deployed and requested flows.
- The **Schedule management** component facilitates flow requests using the flow management and network topology components. It can operate in various

modes to enable or disable flexibility aspects and enforce specific scheduling mechanisms.

- **Flexibility aspects** and **scheduling** options include:
  - Path selection strategies:
    1. Shortest path
    2. Basic flexcurve
    3. Deadline-aware flexcurve
    4. Least path utilization
    5. Random path
  - Flexcurve value computation using basic and deadline-aware formulations.
  - Flow scheduling approaches:
    1. Satisfiability modulo theories (SMT) and incremental SMT methods. Refer to Appendix A.1.1 for details.
    2. Random assignments
    3. First-fit heuristics
    4. Algorithms 2 and 3
- The **Hardware Abstraction** components are interfacing between flow management and physical devices when scheduling succeeds. They are also responsible to generate a viable gate control list (GCL) for each affected device.

A.1.1 *SMT Scheduler*

Given a set of flows F, where each flow f_δ has requirements, we need to extend the notation for requirements to allow for constraints between different flows. The period of f_δ is given by h_δ. The number of hops of f_δ is given by m_δ. The frame size (slot requirements) of f_δ is given by c_δ. The path is given by P^δ.

The deadline requirements (maximum end-to-end delay allowed) for flow f_δ are given by d_δ. Flows may require a minimum traversal time in the network; for this reason, the latency requirement (minimum delay required) for flow f_δ is given by l_δ.

The primary variables for this SMT formulation are the scheduled times for a flow $f_\delta \in F$ at port p_ω. For ease of readability, we extend the notation for assignments A within the port schedule, to encompass different flows f_δ.

$$A^\delta_\omega \in \{0, \dots, h-1\}$$

and the queue assignment for $f_\delta \in F$:

$$Q^\delta_\omega \in \{1, \dots, 8\}$$

Specifying the distance between flow instances

The time between frames of a flow, where the period is less than the hyperperiod h, need to be aligned according to their specified period h_δ. The hyperperiod h, is the least common multiple (LCM) of all flow cycle durations. $n^\delta_{\text{subflows}}$ is the number of subflows for flow δ, calculated as $\left\lceil \frac{h}{h_\delta} - 1 \right\rceil$.

We add secondary variables for each subflow sf of f_δ:

$$A^{\delta,sf}_\omega \in \mathbb{N}$$

The constraints for aligning subflows at the talker and listener are formulated as follows, $\forall f_\delta \in F, \forall sf \in \{1, \dots, n^\delta_{\text{subflows}}\}$:

$$A^{\delta,sf}_1 = A^\delta_1 + h_\delta \times sf, \tag{A.1}$$

$$A^{\delta,sf}_{m_\delta} = A^\delta_{m_\delta} + h_\delta \times sf \tag{A.2}$$

With (A.1) aligning the subflow start time at the talker, and (A.2) aligning the subflow start time at the last port, these constraints ensure that the start and end times of each subflow are exactly one cycle duration apart from the original flow, thus aligning them throughout the schedule. However, the subflows are permitted to be misaligned between talker and listener ports.

Finally, we incorporate subflows into the original variables A^δ_ω, and consider them as initial flows in all subsequent constraints.

Limiting the End-to-End Delay

We limit the end-to-end delay $T(A)$ with port-consecutive delay $t(n, \kappa)$

$$T(A) = c + \sum_{\delta=1}^{m-1} t(a_\delta, a_{\delta+1}) \tag{4.19}$$

for each flow:

$$\forall f_\delta \in F, \quad T(A^\delta_\omega) \leqslant d_\delta,$$
$$T(A^\delta_\omega) \geqslant l_\delta.$$

Alternatively, we can use the simpler notation. However, using this formulation disallows schedule wraps:

$$\forall f_\delta \in F, \quad A^\delta_{m_\delta} + c_\delta - A^\delta_1 \leqslant d_\delta,$$
$$A^\delta_{m_\delta} + c_\delta - A^\delta_1 \geqslant l_\delta.$$

Limit range of scheduled time

Ensure that the scheduled time lies within the hyperperiod h for each flow at each hop:

$$\forall f_\delta \in F, \forall p_\omega \in P^\delta : \quad A^\delta_\omega \geqslant 0,$$
$$A^\delta_\omega + c_\delta \leqslant h,$$

Limit queue assignments

Ensure that the queue assignments are in valid ranges:

$$\forall f_\delta \in F, \forall p_\omega \in P^\delta: \quad \begin{aligned} Q^\delta_\omega &\geqslant 1, \\ Q^\delta_\omega &\leqslant 8, \end{aligned}$$

Fix the queue identifier along the path:

$$\forall f_\delta \in F, \forall \omega \in \{1, \dots, m_\delta - 1\}: \quad Q^\delta_\omega = Q^\delta_{\omega+1}$$

Conflict free schedules

We need to ensure flows are not scheduled at the same time and overlap within port schedules. The set OV^p includes all flows that traverse through the port p.

Next, we define non-conflict constraints for all contended ports, to ensure scheduled frame instances do not overlap. The starting time is either before or after each flow:

$$\forall p, \forall f_\delta \in OV^p, \forall f_j \in OV^p, \delta \neq j: \quad A^\delta_p \geqslant A^j_p + c_j \vee A^j_p \geqslant A^\delta_p + c_\delta$$

To create viable schedules for TSN GCLs, we need to ensure flow isolation. That means queues need to be separate when flows occupy the port at the same time, $\forall p, \forall f_\delta \in OV^p, \forall f_j \in OV^p, \delta \neq j:$

$$\begin{aligned} &\text{Arrival}(A^\delta_p) \leqslant A^j_p \wedge \text{Arrival}(A^\delta_p) \geqslant \text{Arrival}(A^j_p) \vee \\ &\text{Arrival}(A^j_p) \leqslant A^\delta_p \wedge \text{Arrival}(A^j_p) \geqslant \text{Arrival}(A^\delta_p) \Rightarrow Q^\delta_\omega \neq Q^j_\omega \end{aligned}$$

When flows occupy the same queue at the port, ensure that frames arrive only after the other has left the queue already, $\forall p, \forall f_\delta \in OV^p, \forall f_j \in OV^p, \delta \neq j:$

$$Q^\delta_\omega = Q^j_\omega \Rightarrow \text{Arrival}(A^\delta_p) \geqslant A^j_p + c_j \vee \text{Arrival}(A^j_p) \geqslant A^\delta_p + c_\delta$$

Incremental scheduler

To support incremental scheduling, the currently active flows can be included as constants. Conflicts within each constant do not need to be considered. This reduces the number of constraint significantly.

We implemented this scheduler using the Python API of Z3 [97].

A.2 LEMMAS AND PROOFS

Lemma A.1. *For a path* $P = (p_1, \dots, p_m)$*, for all frame sizes* $\forall c \in \{1, \dots, h\}$*, the worst-case computational complexity of the complete basic flexcurve* $b_P(c)$ *given as*

$$b_P(c) = \min_{p \in P} \sum_{\tau=0}^{h-c} 1_{\{C_p(\tau+c)-C_p(\tau)=c\}} \tag{4.2}$$

using the cumulative capacity $C_p(n)$ *with* $\mathcal{T}_{p,\beta}$ *as constant time operation, given as*

$$C_p(n) = \sum_{\beta} 1_{\{n \geqslant \mathcal{T}_{p,\beta}\}} \tag{4.1}$$

is $O(mh^3)$.

Proof. Given the flexcurve calculation of (4.2), we list the computational complexity of all nested operations. We are assuming a constant time lookup for $\mathcal{T}_{p,\beta}$. $\mathcal{T}_{p,\beta}$ gives the time point of the β-th free slot at the schedule for port p.

1. $b_P(c)$ is applied h times, since we need to reflect all frame sizes $\forall c \in \{1, \dots, h\}$, hence $O(h)$.
2. The minimum value is taken for each $p \in P$, hence, $O(m)$.
3. To count the number of arrangements, we aggregate the sum $\sum_{\tau=0}^{h-c} 1_{\{\cdot\}}$. Hence, $O(h)$.
   - With $\forall c \in \{1, \dots, h\}$ from (1.), there are

     $$\overbrace{\underbrace{(h-1+1)}_{c=1} + \underbrace{(h-2+1)}_{c=2} + \dots + \underbrace{(h-h+1)}_{c=h}}^{h \text{ times}} = \frac{h(h+1)}{2}$$

     iterations in total. There are h calls from (1.). Each call of the aggregation yields a linear runtime: $\frac{\frac{h(h+1)}{2}}{h} = \frac{h}{2} + \frac{1}{2}$, resulting in $O(h)$ for each call.
4. $C_p(n)$ is used twice within the indicator term, each $C_p(n)$ aggregates the sum $\sum_{\beta=1}^{h} 1_{\{n \geqslant \mathcal{T}_{p,\beta}\}}$ in the worst-case, i. e., an empty schedule. This requires $2h$ iterations, hence $O(h)$.
5. Each access of $\mathcal{T}_{p,\beta}$ is constant, hence $O(1)$.

The overall worst-case running time of calculating the flexcurve, as in (4.2), can be deduced by combining the complexities of each nested operation. Hence, the total complexity results in $O(mh^3)$. □

Lemma A.2. *The gap-local flexcurve*

$$\tilde{b}_{\Delta}(c) = \max\{0, \Delta - c + 1\} \tag{4.9}$$

is equal to a basic flexcurve at one port:

$$b_{\{p\}}(c) = \min_{p \in \{p\}} \sum_{\tau=0}^{h-c} 1_{\{C_p(\tau+c) - C_p(\tau) = c\}} \tag{4.2}$$

with hyperperiod $h = \Delta$*, when the schedule* s_p *is empty, i. e.,* $\mathcal{T}_{p,\beta} = k$*, for all* $c \in \{1, ... \Delta\}$.

Proof. Given that only one port is affected, we can omit the minimum, therefore

$$b_{\{p\}}(c) = \min_{p \in \{p\}} \sum_{\tau=0}^{h-c} 1_{\{C_p(\tau+c) - C_p(\tau) = c\}} = \sum_{\tau=0}^{h-c} 1_{\{C_p(\tau+c) - C_p(\tau) = c\}}$$

The hyperperiod $h = \Delta$:

$$b_{\{p\}}(c) = \sum_{\tau=0}^{\Delta-c} 1_{\{C_p(\tau+c) - C_p(\tau) = c\}}$$

Because, the schedule is empty, the increase in cumulative capacity over the interval τ to $\tau + c$ always matches c. Hence, the indicator function is always true:

$$b_{\{p\}}(c) = \sum_{\tau=0}^{\Delta-c} 1$$

There are $\Delta - c + 1$ iterations in the sum. The sum counts its own number of iterations. Hence,

$$b_{\{p\}}(c) = \Delta - c + 1 = \tilde{b}_{\Delta}(c) = \max\{0, \Delta - c + 1\}$$

for $c \in \{1, \ldots, \Delta\}$: This matches canonical flexcurve $\tilde{b}_{\Delta}(c)$. Values of $c > h$ are undefined in $b_P(c)$.

□

Lemma A.3. *Any value of a deadline-aware flexcurve is smaller or equal to a basic flexcurve:*

$$b_P^d(c) \leqslant b_P(c) \tag{4.20}$$

$$b_P^d(c) = b_P(c) \text{ if } d \geqslant \max(T) \tag{4.21}$$

Proof. The following proof is cited verbatim from [28], adjusted to fit the notation of this thesis.

The number of admissible assignments in a set of all possible assignments $A \in \mathbf{A}$ for a stream of size c and path P is given by the flexcurve in (4.2). Given a stream deadline $d \geqslant 0$, the number of admissible assignments in $\mathbf{A}$ reduces to the number of assignments in $\mathbf{A}'$ by removing the assignments in the subset $A \in \overline{\mathcal{A}} \subseteq \mathbf{A}$ for which $T(A) > d$, i.e., the deadline is not met. Hence, $\mathbf{A}' = \mathbf{A} \setminus \overline{\mathcal{A}}$. Since $\overline{\mathcal{A}}$ is a subset of $\mathbf{A}$, (4.20) holds as the number of assignments never increases with the deadline $d \geqslant 0$.

If the deadline d is fixed larger than the worst case delay, i.e., $d \geqslant \max(T)$, then $\overline{\mathcal{A}} = \emptyset$, because $D(A) \not> \max(T)$ for all assignments $A \in \mathbf{A}$. Any arbitrary delay of specific assignments can never be greater than the maximum delay. Therefore the number of admissible assignments does not decrease and (4.21) holds. □

A.3 ADDITIONAL FIGURES

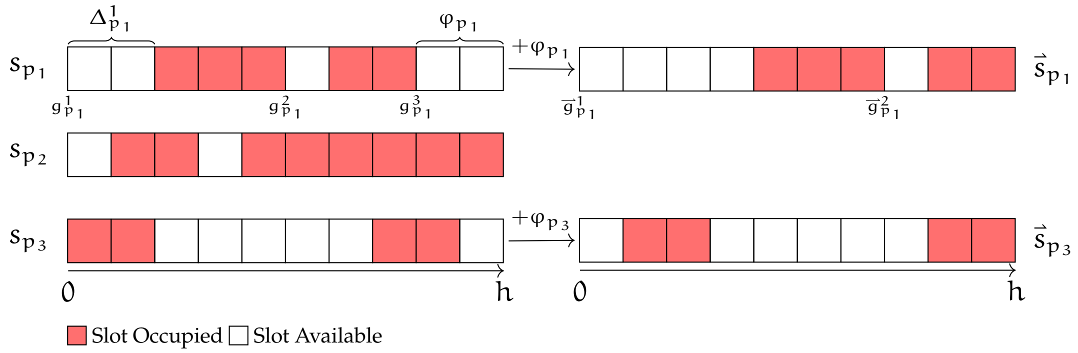


Figure A.1: Consider three cases for gaps in the end-shifted schedule $\vec{s}_p$. The special empty schedule is excluded from consideration. Case s_{p_1} involves a gap at both the start and end of the schedule. Case s_{p_2} features no gap at the end. Case s_{p_3} presents a gap at the end and no gap at the start. For s_{p_1}, the total number of gaps decreases by one as the initial gap is extended. In s_{p_2}, there is no alteration since the schedule remains unshifted. In s_{p_3}, the sequence of gaps is reordered by inserting the final gap at the beginning.

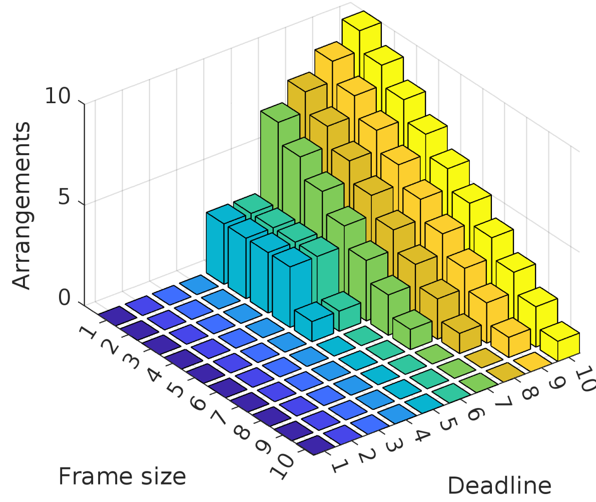


Figure A.2: A deadline-aware flexcurve reflects flexibility with respect to deadline and frame size parameters. As the constraints become increasingly strict, the flexibility value decreases or remains the same.

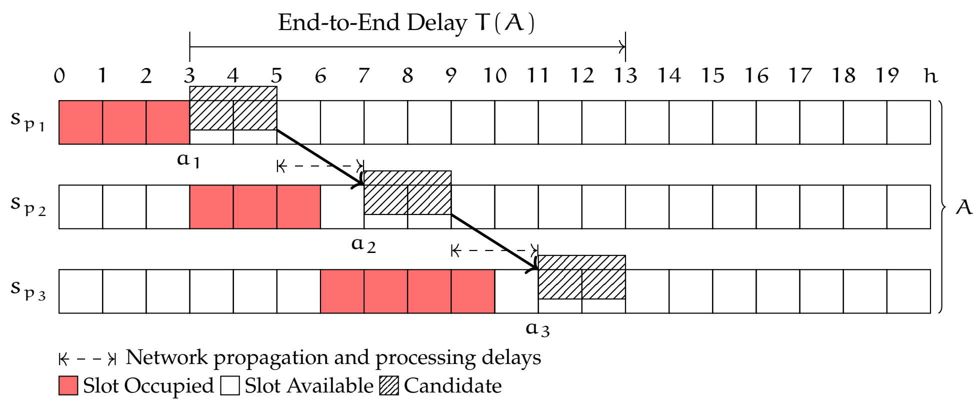


Figure A.3: To consider network and propagation delays in Algorithm 2, the extension of the earliest time a proceeding assignment (a_2, a_3) can be scheduled is required.

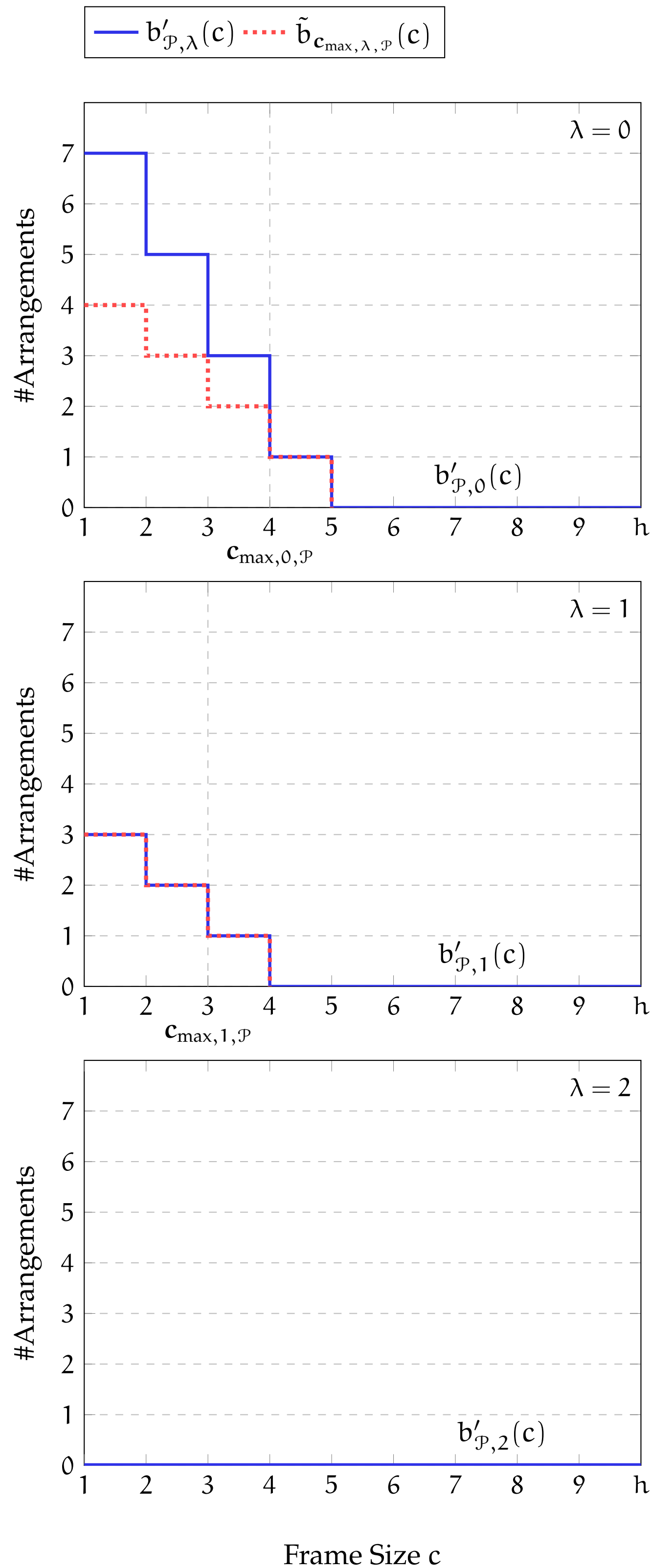


Figure A.4: Large version of Figure 4.6. An example of flexcurve disaggregations is depicted. The initial flexcurve (left, solid) can be disaggregated twice until the residual capacity is exhausted. At each step, the canonical flexcurve (dotted) is subtracted, reducing the capacity by one maximum sized embedding.

A.4 IN-PLACE SCORING SCENARIOS

Scenario 1

Sequence of actions:

- 20 Times: Add Flow Path A
- 5 Times: Add Flow Path B
- 10 Times: Remove Random Flow
- 5 Times: Add Flow Path A
- 5 Times: Add Flow Path B
- 10 Times: Remove Random Flow
- 13 Times: Add Flow Path A
- 20 Times: Remove Random Flow
- 5 Times: Add Flow Path A
- 5 Times: Add Flow Path B

Scenario 2

- 27 Times:
  - 3 Times: Add Flow
  - 3 Times: Remove Random Flow
  - Add Flow Path B
  - Add Flow
  - Add Flow Path B
  - 3 Times: Remove Random Flow
- 3 Times: Add Flow
- 3 Times: Remove Random Flow
- Add Flow Path B
- Add Flow
- Add Flow Path B
- 3 Times: Remove Random Flow

A.5 LIST OF ACRONYMS

API Application Programming Interface

ATS Asynchronous Traffic Shaping

BE best-effort

CBS Credit-based Shaper

CNC Centralized Network Configuration

CUC Centralized User Configuration

FIFO first-in-first-out

FRER Frame Replication and Elimination for Reliability

GCL gate control list

HAL hardware abstraction layer

IIoT Industrial Internet of Things

ILP integer linear programming

PCP Priority Code Point

PIFO push-in-first-out

PLC programmable logic controller

QoS Quality of Service

RDA residence delay aggregation

SDN Software-defined Networking

SMT satisfiability modulo theories

ST scheduled traffic

TAS Time Aware Shaper

TSN Time-Sensitive Networking

B SUPERVISED STUDENT THESES

[1] Marvin Härdtlein. "Hybrid Switch: Dynamic Software Switch Flow Rule Offloading on High Performance Networking Hardware." Secondary supervisor. KOM-M-0715. Master Thesis. TU Darmstadt, 2020.

[2] Ke Pan. "Multi-Path Scheduling in Time-Sensitive Networks." KOM-M-0736. Master Thesis. TU Darmstadt, 2021.

[3] Miron Abraha. "Scheduling Optimization for Time Aware Shaper in Time Sensitive Networks." KOM-M-0747. Master Thesis. TU Darmstadt, 2022.

[4] Patrick van Halem. "Implementation and Performance Evaluation of Asynchronous Traffic Shaping on Real-Time Processors." KOM-M-0738. Master Thesis. TU Darmstadt, 2022.

[5] Tewodros Kebede. "Traffic-Shaping Mechanisms Coordination in Time Sensitive Networks." KOM-M-0757. Master Thesis. TU Darmstadt, 2023.

C

AUTHOR'S PUBLICATIONS

MAIN PUBLICATIONS

[1] Christoph Gärtner, Amr Rizk, Boris Koldehofe, René Guillaume, Ralf Kundel, and Ralf Steinmetz. "Fast incremental reconfiguration of dynamic time-sensitive networks at runtime." In: *Journal of Computer Networks* 224 (2023), p. 109606. DOI: https://doi.org/10.1016/j.comnet.2023.109606.

[2] Christoph Gärtner, Amr Rizk, Boris Koldehofe, Rhaban Hark, René Guillaume, and Ralf Steinmetz. "Leveraging Flexibility of Time-Sensitive Networks for dynamic Reconfigurability." In: *2021 IFIP Networking Conference (IFIP Networking)*. IEEE, 2021, pp. 1–6. DOI: 10.23919/IFIPNetworking52078.2021.9472834.

[3] Christoph Gärtner, Amr Rizk, Boris Koldehofe, René Guillaume, Ralf Kundel, and Ralf Steinmetz. "On the Incremental Reconfiguration of Time-sensitive Networks at Runtime." In: *2022 IFIP Networking Conference (IFIP Networking)*. IEEE, 2022, pp. 1–9. DOI: 10.23919/IFIPNetworking55013.2022.9829815.

[4] Christoph Gärtner, Amr Rizk, Boris Koldehofe, Rhaban Hark, René Guillaume, Ralf Kundel, and Ralf Steinmetz. "POSTER: Leveraging PIFO Queues for Scheduling in Time-Sensitive Networks." In: *2021 IEEE International Symposium on Local and Metropolitan Area Networks (LANMAN)*. IEEE, 2021, pp. 1–2. DOI: 10.1109/LANMAN52105.2021.9478796.

[5] Christoph Gärtner, Amr Rizk, Boris Koldehofe, René Guillaume, Ralf Kundel, and Ralf Steinmetz. "Enhancing Flexibility for Dynamic Time-Sensitive Network Configurations." In: *Proceedings of the 3rd KuVS Fachgespräch "Network Softwarization"*. 2022, pp. 1–2. DOI: 10.15496/publikation-67440.

[6] Chengbo Zhou, Christoph Gärtner, Amr Rizk, Boris Koldehofe, Björn Scheuermann, and Ralf Kundel. "RDA: Residence Delay Aggregation for Time-Sensitive Networking." In: *Proceedings of 2024 IEEE Network Operations and Management Symposium (NOMS 2024)*. 2024. DOI: 10.1109/NOMS59830.2024.10574998.

CO-AUTHORED PUBLICATIONS

[7] Ralf Kundel, Christoph Gärtner, Manisha Luthra, Sukanya Bhowmik, and Boris Koldehofe. "Flexible Content-based Publish/Subscribe over Programmable Data Planes." In: *NOMS 2020 - 2020 IEEE/IFIP Network Operations and Management Symposium*. 2020, pp. 1–5. DOI: 10.1109/NOMS47738.2020.9110381.

[8] Ralf Kundel, Nehal Baganal Krishna, Christoph Gärtner, Tobias Meuser, and Amr Rizk. "Poster: Reverse-Path Congestion Notification: Accelerating the Congestion Control Feedback Loop." In: *2021 IEEE 29th International Conference on Network Protocols (ICNP)*. IEEE, 2021, pp. 1–2. DOI: 10.1109/ICNP52444.2021.9651961.

[9] Bochra Boughzala, Christoph Gärtner, and Boris Koldehofe. "Window-Based Parallel Operator Execution with in-Network Computing." In: *Proceedings of the 16th ACM International Conference on Distributed and Event-Based Systems*. DEBS '22. Association for Computing Machinery, 2022, pp. 91–96. DOI: 10.1145/3524860.3539804.

DEMO PAPERS

[10] Christoph Gärtner, Amr Rizk, Boris Koldehofe, René Guillaume, Ralf Kundel, and Ralf Steinmetz. "Demo: Flexibility-Aware Network Management of Time-Sensitive Flows." In: *Proceedings of the ACM SIGCOMM 2023 Conference*. ACM SIGCOMM '23. Association for Computing Machinery, 2023, pp. 1176–1178. DOI: 10.1145/3603269.3610869.

D

ERKLÄRUNGEN LAUT PROMOTIONSORDNUNG

§ 8 Abs. 1 lit. d PromO

Ich versichere hiermit, dass von mir zu keinem vorherigen Zeitpunkt bereits ein Promotionsversuch unternommen wurde. Andernfalls versichere ich, dass der promotionsführende Fachbereich über Zeitpunkt, Hochschule, Dissertationsthema und Ergebnis dieses Versuchs informiert ist.

§ 9 Abs. 1 PromO

Ich versichere hiermit, dass die vorliegende Dissertation, abgesehen von den in ihr ausdrücklich genannten Hilfsmitteln, selbstständig und nur unter Verwendung der angegebenen Quellen verfasst wurde. Weiterhin versichere ich, dass die "Grundsätze zur Sicherung guter wissenschaftlicher Praxis an der Technischen Universität Darmstadt" sowie die "Leitlinien zum Umgang mit digitalen Forschungsdaten an der TU Darmstadt" in den jeweils aktuellen Versionen bei der Verfassung der Dissertation beachtet wurden.

§ 9 Abs. 2 PromO

Ich versichere hiermit, dass die vorliegende Dissertation bisher noch nicht zu Prüfungszwecken gedient hat.

Darmstadt, 7. Mai 2024

Christoph Gärtner

COLOPHON

This document was typeset using the typographical look-and-feel `classicthesis` developed by André Miede. The style was inspired by Robert Bringhurst's seminal book on typography "*The Elements of Typographic Style*". `classicthesis` is available for both LaTeX and LyX:

https://bitbucket.org/amiede/classicthesis/

Happy users of `classicthesis` usually send a real postcard to the author, a collection of postcards received so far is featured here:

http://postcards.miede.de/

Final Version as of August 22, 2024 (`classicthesis`).