TU WIEN Informatics

TSN Scheduling with Re-Configuration for Dynamic Industrial Networks

BACHELOR'S THESIS

submitted in partial fulfillment of the requirements for the degree of

Bachelor of Science

in

Computer Engineering

by

Sebastian Seitner
Registration Number 01429061

to the Faculty of Informatics

at the TU Wien

Advisor: Univ.Prof. Dipl.-Ing. Dr.techn. Wolfgang Kastner
Assistance: Projektass. Dipl.-Ing. (FH) Dieter Etz, MBA
Projektass. Dipl.-Ing. Dr.techn. Thomas Früwirth, Bsc

Vienna, 10th September, 2023

Sebastian Seitner	Wolfgang Kastner

Erklärung zur Verfassung der Arbeit

Sebastian Seitner

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

Wien, 10. September 2023

Sebastian Seitner

Acknowledgements

First and foremost, I want to express my sincerest gratitude to my wife Lisa. She always supported me during my studies by either giving me the time I needed or by helping to motivate me. There have been times were I was not available much, because finishing your studies and working a full time job consumes a lot of time, so this warrants another huge ‘Thank You‘. I am also grateful that she took the time to proofread this thesis multiple times. She also provided great writing and structural suggestions to improve this work even further. Secondly, I want to thank my advisors Dieter Etz and Thomas Frühwirth for guiding me through the whole process and for providing valuable feedback. I also want to thank them for providing me with the time that was necessary to complete this thesis without pressure, and for keeping up with my irregular schedules.

Kurzfassung

Durch immer weiter steigende Flexibilitätsanforderungen in modernen industriellen Netzwerken entstehen laufend neue Herausforderungen für die zugrundeliegenden Technologien, deren Entwicklung und Verwaltung. Das gilt auch für den Fall des IEEE Time Sensitive Networking (TSN) Standards und dessen Implementierungen. Durch den Umstand, dass die Berechnung von gültigen TSN Schedules ein NP-vollständiges Problem darstellt, ist es nötig, Strategien zu erarbeiten, welche zu lange Laufzeiten nach Änderungen der Netzwerktopologie vermeiden. Andernfalls könnte es sein, dass ein Netzwerk mehrere Stunden nicht korrekt nutzbar ist.

In dieser Arbeit fokussieren wir uns darauf, einen bereits publizierten heuristischen Ansatz zur Berechnung von TSN Schedules, zu erweitern. Ziel ist die rasche Anpassung der bisherigen Konfiguration nach Topologieveränderungen. Wir stellen eine Proof-of-Concept Implementierung vor und präsentieren einige Evaluierungen, um die Anwendbarkeit zu zeigen. Für die Auswertung wurde ein Netzwerkgraph herangezogen, welcher ein kleines bis mittleres Netzwerk darstellen soll. Innerhalb dieses Netzwerks wurde eine unterschiedliche Anzahl an Nachrichten definiert, welche durch dieses gesendet werden sollen. Die Evaluierungsergebnisse zeigen, dass gewisse Parameter, beispielsweise die Zeit mit der sich eine Nachricht wiederholt, einen signifikanten Einfluss auf die Laufzeit des Algorithmus haben können. Wir schlagen deshalb vor, diese Umstände bereits in der Designphase zu berücksichtigen, um bessere Laufzeiten erreichen zu können.

Abstract

The ever increasing flexibility requirements of modern industrial networks create new and complex challenges for the underlying technologies, their development and management. This is also the case for the IEEE Time Sensitive Networking (TSN) standard and its implementations. As the task of finding valid TSN schedules is an NP-complete problem, it is necessary to come up with strategies to avoid prohibitively long computation times when changes in the network happen because this could render a network useless for multiple hours.

In this work, we focus on enhancing an existing and published heuristic approach to enable recalculations in response to topology changes on existing network schedules. The goal is to decrease the time it takes to reach a valid TSN network schedule. We provide a proof-of-concept implementation and run some evaluations to judge the suitability of this approach. The evaluations are based on a network graph which resembles a small to medium sized network. Across this network, we synthesized a varying number of messages to be sent across the nodes. The evaluation results show that some parameters, e.g. the selection of periods within which messages are repeated, which are chosen during the design phase of the network, can have significant impacts on the computation times of the algorithm. Therefore, we suggest that those facts should be considered during the design phase as good as possible to achieve faster schedule computations.

Contents

CHAPTER 1

Introduction

In the past, a lot of different communication protocols for the industrial environment have emerged. One of the earliest communications with a 'point-to-point' connection is RS232 which was standardized in 1960 and still sees some usage today. The first PLC, the 'Modicon 048' emerged in 1968 and about 11 years later, in 1979, Modbus was registered as a communication standard by Modicon. Also in the late 70's Ethernet was born which gained popularity over time, especially in Local Area Network (LAN) environments in the Information Technology (IT) area. During the 1980s, fieldbus systems gained traction and a lot of different approaches emerged in a relatively short amount of time, and the standardization efforts led to the so-called Fieldbus War [FS02]. An example of a fieldbus system which was part of this is PROFIBUS. Since about 20 years the usage of so-called industrial Ethernet is getting a lot of attention and many modern communication approaches use Ethernet as their basis. A few examples include Profinet (2002), EtherCAT (2005) and Time Sensitive Networking (TSN).

In general, these protocols enable machines to communicate effectively and reliably to, for example, link control systems to their corresponding sensors and actuators or let machines exchange data between each other. They operate in the so-called Operational Technology (OT) realm. The purpose of OT is to use hardware and software to measure and control industrial equipment and how it interacts with the physical world. In short, they are the technologies which keep an industrial plant running or enable it to operate in the first place. This also encompasses the fact that the communication has to be predictable and very reliable even in hostile environments where lots of interferences can happen. Many industrial communication protocols have the major disadvantage that they are usually different from vendor to vendor and are, therefore, incompatible to each other. This leads to so-called vendor lock-in, where a customer is bound to buy equipment from a single manufacturer even if the device does not have the best

price-performance ratio for the specific use case. Therefore, it causes slower innovation, lower flexibility and hierarchical systems. Those hierarchies are formed by the need of placing gateways, which translate protocols, between different parts of the network.

TSN is an open suite of standards which solves this issue by being compatible between different manufacturers. Therefore, TSN allows more flexibility in selecting devices and also in the overall network architecture because no gateway devices are necessary to translate between different protocols. Additionally, the standards are not developed by a single company. In fact, there are multiple companies driving the development. The basis for TSN is the already existing Ethernet standard, more specifically IEEE802.1Q. Ethernet is a good base because it is widely used, cost effective, fast, scalable and well understood ([Gmb21]). TSN is not the only industrial communication protocol building on top of standard Ethernet—EtherCAT and PROFINET are just two other examples ([Com20]). One essential property that standard Ethernet protocols cannot provide is guarantees regarding the transmission times and latencies. Therefore, the communication is non-deterministic, which is not a problem for normal IT usage, but for industrial communication such guarantees are necessary to enable real-time data traffic.

Real-time in this context means that messages must arrive at their destination within certain time limits. This is, for example, crucial in control systems to guarantee proper operation. For a simple example, one can imagine a conveyor belt that has to stop when a product reaches a light barrier. If the time it takes to transmit the message from the light barrier to the conveyor belt motor controller is not guaranteed, then also the position, at which the product stops, is undefined. Therefore, it matters whether the message takes 10 milliseconds or 1 second to reach its destination. Real-time does not necessarily imply that a data transmission has to be fast, but it has to be within certain guaranteed bounds. In some cases, those time limits may make the difference between smooth operation and catastrophic failure.

Traditionally, the communication networks within machines and on the factory floor in general are more or less decoupled from the existing IT networks. This hinders the ability to automatically gather data from machines. Having easy access to machine-internal data from the IT network would open up a lot of new possibilities. Processes could be analyzed and automatically improved, or preventive maintenance could be applied to reduce downtimes.

In the emerging 4th Industrial Revolution, one of the core concepts is the convergence of the OT, and the IT networks. This obviously presents various challenges ranging from security down to the necessary real-time requirements in the OT realm. TSN enables the IT and the OT networks to share the same physical network because both can use standard Ethernet. On such converged networks, the traffic of both networks needs to be able to coexist. TSN has mechanisms to prioritize real-time traffic and send low priority traffic only if it does not disturb real-time communications. Because both networks share the same cables, the wiring effort, the cost and the maintenance burden is reduced. Although this deployment scenario comes with a lot of complexities, it will open up a lot of new possibilities in the future, which will be worth the effort.

Generally, the flexibility requirements in the industry are also increasing drastically, for example due to smaller lot sizes and more variety in production. This also leads to more frequent changes within the applied networks in terms of transmitted messages and also in the connections themselves. This means that the underlying technology, like TSN, needs to be able to adapt to changes and be easily reconfigurable in a short amount of time, while still upholding the strict guarantees for real-time traffic. This work is concerned with those increasing requirements of fast reconfigurability in industrial automation. The focus of this work will be on the (re-) calculation of schedules (when which messages are allowed to be transmitted) which conform to the given network topology and all timing requirements. The possibility to generate such schedules within seconds is a major part in the ability to reconfigure a machine, a network or a whole plant.

In the remainder of this work, we will first give a basic overview of TSN and the TSN scheduling problem with which we are concerned in Chapter 2. In this chapter, we then move on to provide a brief overview of different existing approaches to this challenge. Beginning in Chapter 3, we will start to describe an algorithm to achieve the calculation of such TSN schedules. Furthermore, we will describe some mechanisms we implemented to be able to compute new schedules after changes in the underlying network occur. To conclude this work, we will show a few evaluations we did on our implementation in Chapter 4 and provide an outlook for future possibilities of the algorithm in Chapter 5.

CHAPTER 2

State of the Art

In this chapter, we will introduce the concept of TSN scheduling. Additionally, we will briefly mention a few existing approaches, which tackle this challenge.

The problem of finding a schedule in a TSN network is a current challenge for the widespread adoption of TSN. This issue is amplified by the circumstance that the requirements on industrial networks are becoming increasingly flexible and that the topology cannot be assumed to be static over the lifetime of the setup. This makes the development of fast and flexible algorithms necessary.

The focus of this work is on enabling fast, online (re-) calculations of the schedule if anything in the network changes. This also means that, we are not necessarily interested in finding the optimal solution, but rather to find a feasible solution within a short timesframe.

2.1 Introduction to Time Sensitive Networking (TSN) and TSN Schedules

TSN is an approach to utilize existing Ethernet capabilities for real-time applications with strict timing requirements. The IEEE Time Sensitive Networking Task group is a successor to the previous Audio-Video Bridging (AVB) task group and continues their work under a new name. This change resulted from an expansion of the focus areas of the group. Traditional Ethernet networks are inherently non-deterministic, which means they cannot provide any guarantees about latencies and transmission times. The TSN standard suite is an extension to the existing IEEE 802.1 standards, which are concerned with Ethernet networking and all its components.
As a basis for guaranteeing any timing requirements, a TSN network needs a reliable way of synchronizing clocks within the network. Such an approch is defined in the

IEEE802.1ASrev standard. With this time synchronization method, a global notion of time can be established within the network where all clocks differ at most $\delta\mu$s from each other at any point in time, which gives us a bounded precision for the clock synchronization. For industrial networks, it is desirable to have a value of δ of less than 1μs. An in depth analysis of the synchronization quality in large industrial networks can be found in [GSDP17].

As mentioned, a major goal of TSN is bridging the gap between the OT and IT world. For this reason, it has to support traffic with varying timing requirements—so-called mixed-criticality traffic. There are three broad traffic categories, which we want to distinguish:

- Time-Triggered (TT) with time-aware shaping: Hard real-time

  Traffic for which it might be catastrophic to miss time deadlines

- Audio/Video (AVB) with credit-based shaping: Soft real-time

  Traffic for which an occasional miss of a deadline does not result in a failure

- Best Effort (BE)

  Traffic for which no guarantees in regard to timing exist.

In TSN, a flow is a periodic message, which is sent from A to B, is repeated after its associated period and has to arrive within a certain time (deadline). A flow might consist of multiple frames. If a sender wants to transmit a message of 4500 bytes, this results in three Ethernet packets containing the frames, with a size of 1542 bytes each. The maximum size of the data payload, the Maximum Transmission Unit (MTU), for Ethernet is 1500 bytes. The extra 42 bytes stem from the size of the preamble (7 bytes), the start frame delimiter (1 byte), the Ethernet header including 802.1Q VLAN tag (18 bytes), the Cyclic Redundancy Check (CRC) (4 bytes) and a necessary interpacket gap between two packets (12 bytes). Flows always repeat with a defined period, which therefore determines an upper bound for the usable deadlines.

One important concept in many TSN scheduling algorithms is the so-called hyperperiod. Each flow has a defined period in which it is repeatedly sent across the network. The hyperperiod describes the overall period in which a certain schedule repeats itself. Therefore, the hyperperiod is defined as the Least Common Multiple (LCM) of all individual periods. This concept can be applied to the whole network or only to individual parts such as queues and links. In this narrower case the hyperperiod is the LCM of all flows that pass through the specific queue or link.

In terms of TSN, a switch can be viewed as follows and this explanation is aided by Figure 2.1. The port at which a frame arrives at the device is called ingress port. From there the frame goes through the switching fabric where it is decided at which egress port it will leave the switch towards its destination. Each egress port has eight queues associated with it. Every transferred frame is placed into one of those queues based on the three bit wide Priority Code Point (PCP) field in the Ethernet frame header, defined in 802.1Q. All queues have a so-called gate at their output which can control when frames will be sent from a specific queue. The mechanism by which the gate openinges are defined is galled the Gate Control List (GCL). If multiple gates are open at the same time then the frames in queues with a lower index take precedence and are sent first.

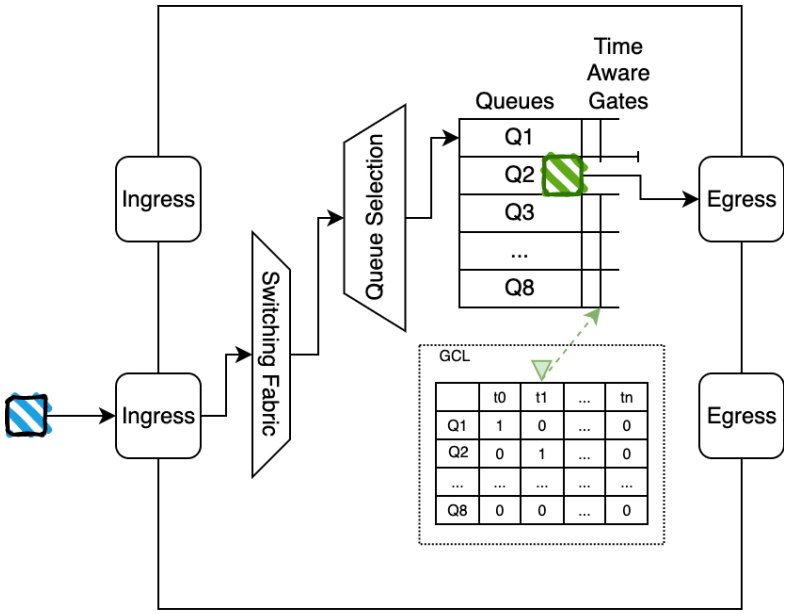

Figure 2.1: Simplified overview of a TSN device

TSN povides two different mechanisms for traffic shaping. One of them is a credit-based shaper (CBS), which can be used e.g. for audio/video streams to achieve low latencies and high link utilization but limit bursts (see [IET13] page 54ff). This traffic class is not suitable for applications with hard real-time requirements. The other one is a time-aware shaper, which is suitable for hard real-time traffic.

In this work, we will only concentrate on the Time-Triggered (TT) traffic category. The other two categories have no influence on the transmission of TT traffic because they will not be allowed to be in flight during time windows that are exclusively reserved for TT flows. In TSN schedules, we can make sure that no other traffic interferes with high priority messages in multiple ways. The simplest, although not the most bandwidth-efficient way, is to include so-called guard bands before the start of critical transmissions. These guard bands have the size of the largest frame which could interfere. Inside the

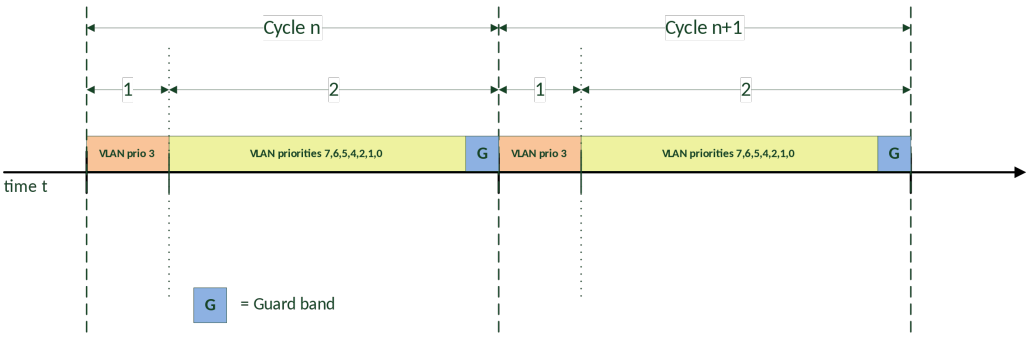

Figure 2.2: TSN scheduling example with guard bands. Source: [Com17b]

guard band no new transmissions are allowed to start. This way, it can be guaranteed that neither the egress port, nor the link is occupied by lower priority traffic once the TT traffic is scheduled to be sent. An example of this can be seen in Figure 2.2.

The other, more complex, solution is frame preemption, which is defined in IEEE 802.1Qbu/802.3br. In this case, a frame, which would interfere with a higher priority transmission, is split, the transmission is stopped, the high priority frame is sent and, after the transfer is finished, the lower priority frame is continued and then reassembled at the receiver, once it has fully arrived. Figure 2.3 shows a representation of this method. The remaining guard band has the purpose to not allow transmissions to start if the preemption overhead would outweigh the gains. This approach is more efficient in using the available bandwidth, but comes with a lot of implementation complexity and is not widely adopted in scheduling algorithms yet, especially the case where the preemption is used on TT frames.

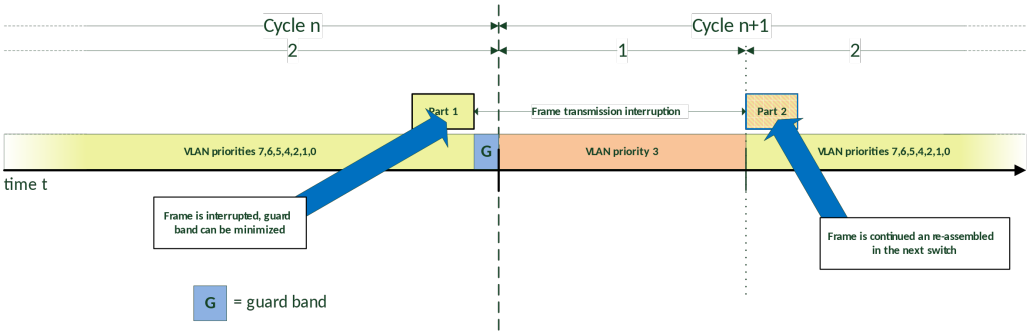

Figure 2.3: TSN scheduling example with frame preemption. Source: [Com17a]

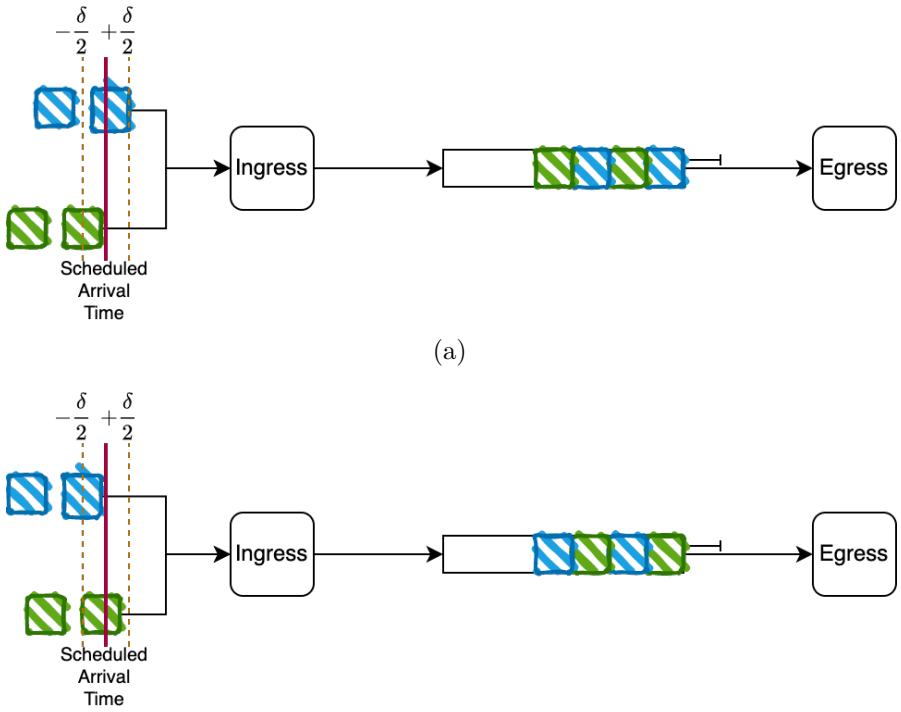

Figure 2.4: Possible non-determinism in queues

Furthermore, the queues of an egress port can be seen as critical sections in the transmission path and care must be taken if multiple flows use the same queue in one device. Due to the imperfect clock synchronization, it might happen that two frames, which are scheduled to arrive at about the same time, behave non-deterministically and could swap their order either way. An example of this can be seen in Figure 2.4. It illustrates a scenario in which the scheduled arrival times of frames from different flows are the same. This leads to the problem that the order on the egress link is not deterministic because the frames can arrive at any time within $\pm\frac{\delta}{2}$ of their scheduled arrival time. To avoid this situation, frames of different flows in the same queue have to be temporally spaced out by at least the distance of δ. Additionally, different traffic types should not be mixed in the same queues to guarantee that they do not interfere with each other. As the number of priorities is limited, it is advisable to only use the minimum number of queues required for TT traffic and to leave more queues free for lower priority traffic (AVB or BE) and, therefore, increase its Quality of Service (QoS).

Based on the availability of synchronized clocks, it is possible to prioritize Time-Triggered traffic with the application of a Time Aware Shaper (TAS). The functionality of the TAS is defined in IEEE 802.1Qbv. This mechanism determines which frames are allowed to be sent across network segments at certain times. In TSN, this prioritization is established by the definition of GCLs. These lists are defined for each queue in all egress ports of the

devices and specify the times at which frames are allowed to be transmitted from which queues. If two different queues are open at the same time, then frames of the queue with the lower index have priority. If the openings of the queues are concerted correctly and are mutually exclusive, it is possible to guarantee deterministic behavior, for hard real-time requirements. For reference, one can look at Figure 2.1 where a simplified overview of a TSN switch is given. Computing all those individual GCLs based on the information about the flows and the topology of the network (including performance information about the existing links) is the scheduling problem we want to solve.

2.2 Related Scientific Work

There exists a multitude of different approaches to the TSN scheduling problem. They range from Tabu-Search algorithms over genetic algorithms to approaches which try to solve the problem to optimality with tools like Integer Linear Programming (ILP) or Satisfyability Modulo Theory (SMT) solvers. Many of the current publications make the assumption that the network is static and that the routes in the system do not change over time or they require a full rescheduling without taking advantage of previous computations (e.g., [SSN19], [RP17], [DN16], [RZCP20]). In situations where the network can indeed be considered static, long running algorithms are feasible as the schedule can be computed offline before the deployment. There are various approaches which solve the scheduling problem to optimality, but as this is an NP-complete problem (a proof can be found in [RP17]) it does not scale well to large industrial networks and might run for hours, or even days, to find a solution. Therefore, it is necessary to choose an approach that balances execution time with solution quality for the given context.

2.2.1 ILP Solver Based

In [DN16], the authors map the TSN scheduling problem to the well-known No-Wait Job-Shop Scheduling problem ([MP02], [MMR99]) and formulate it in terms of ILP. Their evaluation showed that the ILP computation can take an unreasonable amount of time of up to three days for a relatively small problem instance with 50 flows. Although it should be noted that the solver found feasible but not yet optimal solutions within a shorter time window and then improved the solution as the time progressed. The authors of [RP17] also formulate one of their presented solutions in terms of ILP and compare it to their heuristic approach, on which this work is based.

2.2.2 SMT Solver Based

The work of Li et al. [LLJ+20] proposes the use of the existing Z3 SMT solver [Res] to solve the scheduling problem by presenting the solver the according constraints they define in their paper. In contrast to most publications, they do not adopt the concept of using a hyperperiod for all flows, but rather use a base period, first introduced by Nayak in [NDR16], which is essentially the greatest common divisor of all flow periods. This

alleviates the problem of very large hyperperiods and makes the SMT solver approach more tractable. Because, in the case of a large hyperperiod, flows with a small period have to be placed into the schedule more often, which increases the overall runtime.

2.2.3 Heuristic Approaches

The long runtimes of the aforementioned strategies show that it is necessary to develop different solutions which require less computational time and still yield reasonably good schedules. This is especially true for networks that cannot be assumed to remain static. A heuristic algorithm is designed to trade off optimality and completeness for speed. This means that a heuristic approach can be significantly faster but it might result in a non-optimal solution or might not find a feasible schedule, even if one exists. Typically, the heuristic function guides the algorithm through the search space to explore only solutions that seem good for the heuristic. This limits the number of solutions the algorithm needs to consider and, therefore, speeds up the overall process.

2.2.4 Other Approaches

In addition to their work on an ILP approach, the authors of [DN16] also developed a Tabu-Search-based algorithm to compute the schedules. For their evaluation of the Tabu-Search approach, they restricted the maximum runtime of the ILP solver to 300 minutes. Then they compared the solutions and concluded that their proposed Tabu Search algorithm was faster and fared better on average in terms of the solution quality.

CHAPTER 3

Design and Implementation

In the following sections, we want to select a suitable algorithm for rapid TSN scheduling and give an insight into how the chosen algorithm works and how it was implemented. Furthermore, we will mention a few of its limitations and describe some data structures, which were used to solve the problem. The source code for this implementation can be found in the GitLab repository [Sei23].

3.1 Basic Concepts

This work takes the heuristic strategy, developed by Raagaard et al. ([RP17]), as a basis and adds some improvements on top of it. Those additions are mentioned as possible future work in the original publication.

Any algorithm for TSNs schedule calculation takes in the topological definition of a network including all connections (for their attributes see Table 3.1) with their corresponding bandwidth and propagation delays on those links. The second part of the input are all the flows with the necessary metadata, as described in Table 3.2, that should be scheduled. The overall system structure is represented in Figure 3.1.

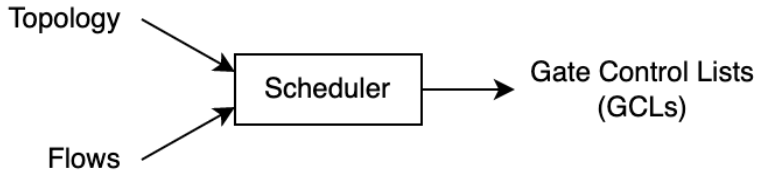

Figure 3.1: Basic system structure

The main functionality of the implementation is the calculation of a feasible schedule in a given network with a heuristic strategy that schedules the flows one-by-one to limit the overall solution space. This approach is selected because it is possible to quickly find a solution in many cases. Obviously, this comes at the cost of forsaking optimality and eventually missing feasible solutions in some cases.

The original strategy proposes to sort the flows according to certain criteria. The idea is to try to schedule the most difficult flows first and add the easier ones later on. How the flows are ordered and what 'difficult to schedule' means will be explained later on. After the initial ordering, the flows are scheduled one-by-one and the algorithms priority is to use as little queues as possible. As a secondary measure, it strives to minimize the end-to-end latency. Therefore, the algorithm tries to schedule the flow into the first queue in all cases. If this is not successful the next queue is tried until it is successful or no more queues are available. In the latter case, the algorithm exits and does not produce a full schedule. The details of how a single flow is scheduled will be explained in the remainder of this chapter.

3.2 Limitations

As a greedy scheduling approach is chosen and the algorithm schedules one flow after the other, it might not find an optimal solution or no solution at all even if one exists. This balances speed with a reasonably high success rate of 90% according to [RP17].

We mainly focus on the options defined in IEEE 802.1Qbv, and assume specific topologies without redundant links, as defined in IEEE 802.1CB, for our use-cases. Another limitation is that the approach does not consider the preemption capabilities of TSN, which are defined in IEEE 802.1Qbu. This would increase success chances in cases with high link utilization where the already scheduled frames leave lots of small gaps in the schedule that are too small to fit full frames. With preemption capabilities, those gaps might also be used.

Furthermore, the algorithm only takes the shortest weighted paths between source and destination of each flow into consideration. Although, it would be fairly straightforward to add the possibility to consider multiple routes of a frame before aborting with no result. This, however, would not consider alternative routes for already scheduled frames.

3.3 Improvements over the Original Algorithm

We added some improvements on top of the original description of the algorithm. Namely, this is advanced bookkeeping of the scheduled flows. We keep track of the time-slots the frames of flows are assigned to. Additionally, there are mechanisms to efficiently remove flows from the schedule. If the underlying topology changes, the affected flows can be quickly removed from the existing schedule and the algorithm is run again with the now unscheduled flows and it tries to fit them into the schedule again. This also

forms a basis for the envisioned delta evaluation of the proposed Greedy Randomized Adaptive Search Procedure (GRASP) metaheuristic approach for schedule optimization from [RP17]. This addition could be used to speed up the neighborhood computations in GRASP and derive better solutions in a shorter amount of time. To enable this, we use a custom implementation of a doubly linked list for storing the time-slots and an index, which keeps track of the association of a flow to all slots it occupies. Both of these structures are described in Section 3.4.2.

3.4 Implementation Details

As the implementation is based on the work of Raagard et al. ([RP17]), we only give a rough overview of the original algorithm and refer the interested reader to the original work which describes the approach in more detail.

3.4.1 Network and Flow Model

The topology of the network, including its metadata, is modeled as a directed graph $\mathcal{G}(\mathcal{L}, \mathcal{N})$. $\mathcal{L}$ is the set of all links connecting nodes of set $\mathcal{N}$ with each other. Each link has the attributes listed in Table 3.1, associated with it. For details about the UtilizationList structure, please refer to Section 3.4.2. Due to the fact that TSN uses the 3-bit wide PCP header field for selecting the queue for a frame, the number of available queues has an upper bound of 8. The attributes which are associated with a flow are shown in Table 3.2.

$s \in \mathbb{R}$	Transmission rate
$d \in \mathbb{R}$	Propagation delay
$sp \in \mathbb{N}$	Source/Egress port number
q	List of UtilizationLists representing egress queues at the source node
u	UtilizationList of the link itself

Table 3.1: Link attributes

$T \in \mathbb{N}$	Period in μs
$D \in \mathbb{N}$	Deadline in μs
$s \in \mathcal{N}$	Source Node
$d \in \mathcal{N}$	Destination Node
$size \in \mathbb{N}$	Total size in bytes

Table 3.2: Flow attributes

3.4.2 Utilization List and Utilization Index

To be able to recalculate a schedule when the topology or the flows change, the algorithm has to keep track of the assignments of flows to time-slots in queues and links at all hops of the route from source to destination. This enables the removal of flows from the schedule without a full recalculation. If a new flow is added, the algorithm can try to schedule it on top of the current solution. If a link is removed from the topology, the system knows which flows are affected by the change and can remove those from the solution and try to schedule them again without touching any other flows.

To enable this functionality, we implement a special form of a doubly linked list which we call UtilizationList. Instances of this list are used to represent every queue and each link in the system. They store the information whether a certain time-slot is free or not. If a frame is assigned to a slot, a reference to the corresponding flow is also stored. On top of this, the algorithm builds an index which creates a mapping from a flow to all slots that it uses.

Utilization List

The UtilizationList is a doubly linked list which represents the utilization of a queue or a link. A node/entry in the list is called slot from here on. A slot represents a slice of time, defined by a start and end time, which can be either free or occupied by one specific frame. The start and end times are non overlapping and there are no gaps between slots, so the list represents one continuous timeline. The list, as a whole, also has a certain period after which all entries reoccur. This value is equivalent to the hyperperiod of all flows that occupy a slot in the list. The implementation keeps track of all individual periods and their count to be able to reduce the hyperperiod if a period is not present anymore because all frames that occured with this period were removed. If this is the case, then the length of the list will be reduced accordingly.

In addition to pointing only to its neighbours, each free slot also has a pointer to the next and previous free slot. This enables faster iterations when searching for free slots. Occupied slots do not possess those pointers because we do not iterate over occupied slots often and, therefore, the increased computational overhead for keeping the references up to date is not beneficial. Furthermore, the utilized slots also store a reference to the flow of which the hosted frame is a part of. This connection is used for some calculations to attain the period of the frame in a particular slot. An example representation of such a list can be seen in Figure 3.2. The top left and top right corners show the respective start and end time of the slot in μs. The center indicates whether a slot is free or occupied. In case of an occupation, the reference to the flow and the particular frame are shown. The lower corners indicate the links to the previous and next free slots. If there is no arrow pointing to this field then this indicates that this link is not populated in the slot. One can see that only free slots have values for those pointers.

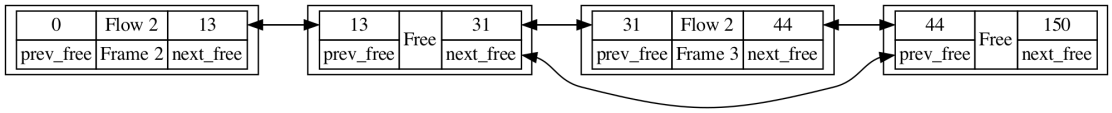

Figure 3.2: Example representation of a UtilizationList

Utilization Index

To keep track of all the associations of flows to the respective slot instances in all the UtilizationLists, we use a dictionary which is keyed by the flow instance and contains a stack of tuples of slot instances and UtilizationList references. With this information, each slot can be be easily obtained to be freed in the corresponding list.

3.4.3 Calculating Shortest Paths between Nodes

The first step in finding a schedule is to determine the route each flow should take through the network. This is done by finding the shortest weighted path through the topology via Dijkstras algorithm [Dij59]. The weight of each link is based on its bandwidth and the propagation delay. If there is more than one path, only the shortest path is considered.

This step can be viewed as a preprocessing step for the rest of the algorithm, because the assigned path is never changed and only a single path per flow is evaluated. We might miss out on some potential solutions but this simplification reduces the overall complexity of the scheduling.

3.4.4 Ordering Flows

After all distances are calculated, the flows are ordered depending on their 'difficulty' to be scheduled. The sorting criteria are as follows:

1. Deadline

   Shorter deadlines come first.

2. Period

   Shorter periods come first.

3. Length of the route

   Longer routes come first.

The reasoning behind this order is that flows with a short deadline are more restricted and harder to schedule successfully. If two frames possess the same deadline, the one with the shorter period comes first because it is repeated more often within the hyperperiod and is therefore harder to fit into the schedule. Flows with a longer period have more options for insertion within the hyperperiod. As a last tiebreaker, the length of the route is considered because it is assumed that flows with a longer route are harder to place as they have to fit into the schedule at more places.

3.4.5 Scheduling Flows

Based on this order, every flow is scheduled one-by-one until all flows are inserted into the schedule or the program fails to schedule a flow within its constraints. In this context, scheduling a flow means that the algorithm searches for an offset ϕ [μs] for each frame of the flow on each hop along its route. This offset determines when the frame should be sent to the next hop. This value is the same for all repetitions of the flow (in one egress port) within the overall hyperperiod. This restriction is in place to further reduce the size of the overall search space and limit the jitter of the flow. As a consequence, the algorithm must find values for ϕ that result in no violations in any repetition. This is one of the reasons why flows with a shorter period are scheduled earlier than others. A value of $\phi = 25\mu$s means that the corresponding frame e.g. $f_{1,2}$ (denoting the second frame of flow 1) is sent at the following times assuming a period ($f_1.T$) of 150μs:

$$\phi, f_1.T + \phi, 2 * f_1.T + \phi, 3 * f_1.T + \phi, \ldots \equiv 25\mu\text{s}, 175\mu\text{s}, 325\mu\text{s}, 475\mu\text{s}, \ldots$$

To determine the sending offset of a frame, it is necessary to check if the frame fits into all UtilizationLists along its path in all repetitions of the flows period. To calculate this, we can 'fold' the UtilizationList A into a new instance B with the flows period which has occupations according to all the slots of the original list. To achieve this, we iterate over the slots in A and calculate start and end times in B based on the overall period of B.

$$\begin{aligned} s &\in A.slots \\ start &= s.start \mod B.period \\ end &= s.end \mod B.period \end{aligned}$$

If *end* is less than *start*, then the slot wraps around the end of B. In this case, we occupy one slot from *start* to $B.period$ and the another one from 0 to *end*. Otherwise, we utilize one slot from *start* to *end*. If all lists along the path are folded together the resulting list contains all the information about the flows path. Based on this folded list the algorithm can search for a free slot in which the current frame fits. An illustration of this process can be seen in Figure 3.3

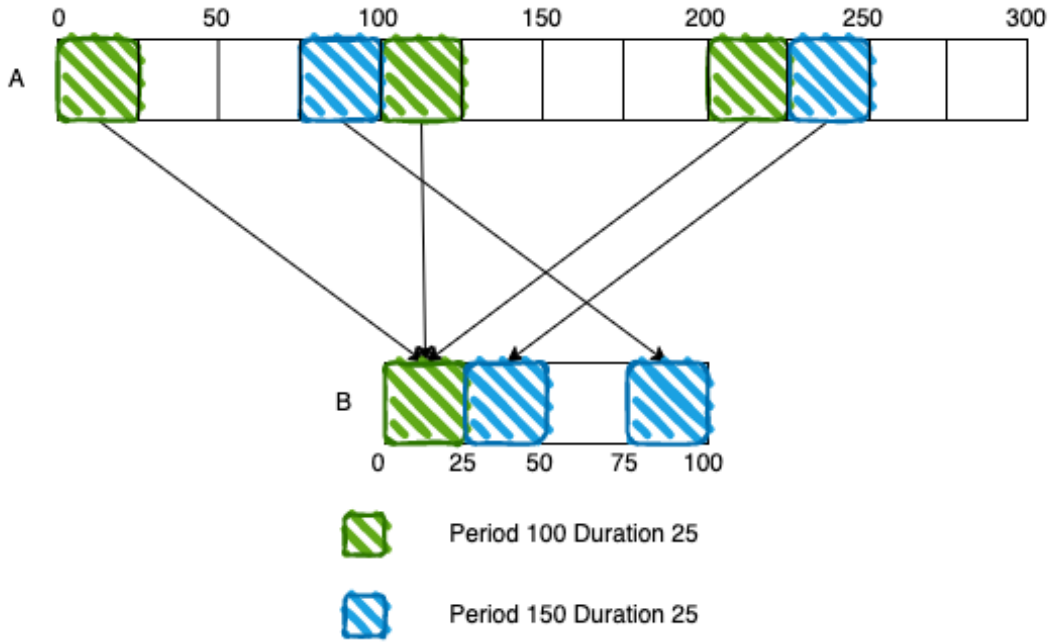

Figure 3.3: Example of folding a UtilizationList

If a flow can not be placed at a certain hop and other queues are available, the algorithm tries to place the flow into another queue. There is an option to have individual queue assignments on each hop or to force the flow to be assigned to the same queue on all nodes in the route. This can be used if the available system is not able to support different queue assignments on a hop to hop basis. In such a case, the queue assignment is changed on all nodes along the way and the flow must be put in the same queue on each hop.

3.4.6 Removing Flows and Reacting to Network Changes

If the underlying topology changes, a recalculation of the schedule can be triggered. The two graphs, representing the old and the new network structure, are compared and all links, that are affected by the change, are determined. With that information, we can determine all flows that need to be removed from the existing schedule. For these flows we can look up the occupied slots in the utilization index (see Section 3.4.2) and free them and mark the flow as unscheduled. Now, the schedule contains only the flows which do not travel across any of the removed links. Then the algorithm handles the scheduling of the affected flows in exactly the same way it would treat a full reschedule, but with only the removed flows. In the case that no valid schedule can be found, the implementation resorts to a full reschedule, which has an increased chance of success, although a longer runtime.

CHAPTER 4

Evaluation

In this chapter, we will describe the evaluation procedure that was used to test and benchmark the implementation. The effort was focused on generating a realistic industrial scenario and evaluate the influence of different parameters on the overall execution time of the algorithm.

4.1 Procedure

The evaluation has two main parts: the graph generation and the flow selection. Those will be detailed in the following sections.

4.1.1 Graph Generation

The first step in the evaluation was to generate a realistic network graph, which will be the same for the rest of the test procedure. But it should be noted that the test procedure allows for arbitrary graphs as input, which enables the evaluation of different scenarios. The graph generation is based on the 'networkx' [HSS08] python library to create a random graph with multiple interconnected components with 50 nodes in total. In this graph, all the edges are set to have a bandwidth of 1Gbps and a negligible propagation delay. Then a minimum spanning tree is calculated to transform it into a valid network topology with no redundant routes. The resulting graph can be seen in Figure 4.1.

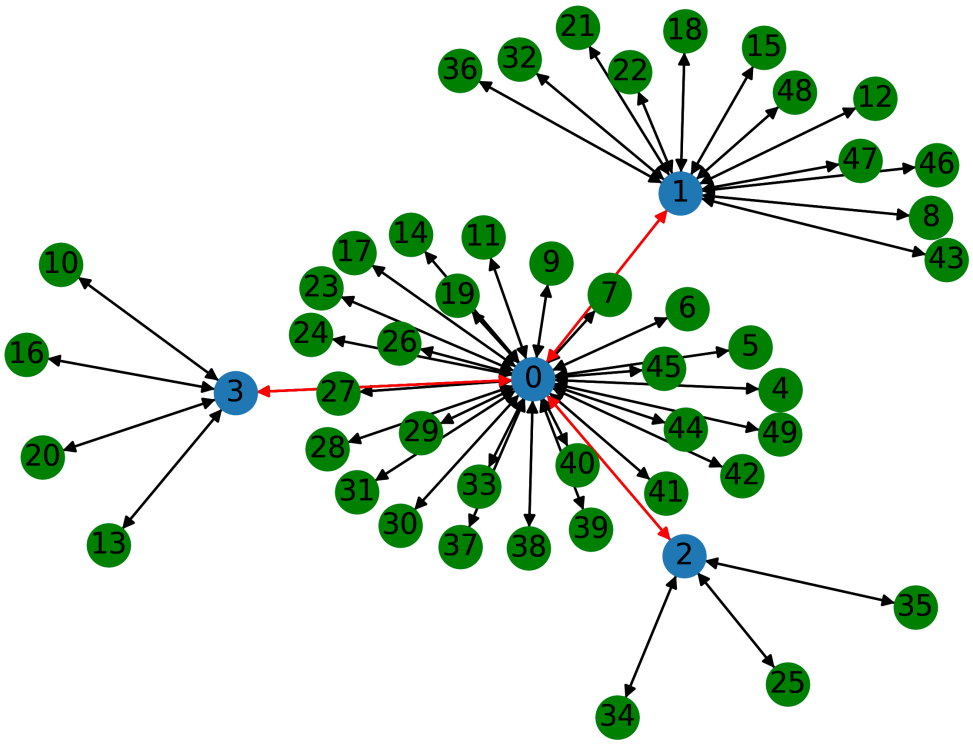

Figure 4.1: Evaluation network graph

4.1.2 Flow Selection

Flows are only sent between end systems. Therefore, the first step is to determine all end systems and group them by their membership to certain network components. This grouping is achieved by detecting all bridges of the graph. Bridges are edges for which it holds that, if they are removed, the graph will break up into multiple connected components. In Figure 4.1, end systems are shown as green nodes and the bridges are shown as red arrows.

For the synthesis of the flows, a few parameters are defined to tweak this generation. These parameters are listed in Table 4.1.

The next step in the selection is to determine the source and destination nodes for the flow. The source component is chosen randomly, but the chances for picking a specific component are proportional to its size (Larger components are more likely to be selected). After that, the destination component is chosen according to *lcc* (if the flow will leave the source component, the other component is again chosen according to its size). Within those two components, the source and destination nodes are picked at random. The flow size is a Gaussian sample taken from the interval [sl, su]. The period of a flow is also

Number of flows n	The total number of flows in the network
Leave Component Chance lcc	A percentage that describes the chance that a flow is sent across one or more bridges into another component
Size Lower Bound sl	The lower bound of the flow size in bytes
Size Upper Bound su	The upper bound of the flow size in bytes
Period Lower Bound pl	The lower bound of the flow period in μs
Period Upper Bound pu	The upper bound of the flow period in μs
Allowed Periods ap	A set of period values which are allowed to be chosen
Number of Distinct Periods ndp	The size of the set ap

Table 4.1: Parameters for flow selection

a Gaussian sample from the interval [pl, pu], but then mapped to the nearest value in the ap set. This value is used for both the period and the deadline of the flow for this evaluation.

4.1.3 Selecting Allowed Periods

To determine the set of allowed periods ap, we opt for a simple greedy approach which picks values from the given range [pl, pu] in such a way that it keeps the LCM low. The algorithm takes the range of periods and the desired number of distinct values ndp as parameters. It first selects the lower and upper bound of the range. Then, it checks all remaining possible values and groups them into sets according to the resulting LCM if they were added to the current selection. If the current LCM of the set will not increase by the addition, the value is added to the result set immediately. After that, the largest groups are selected. From this pool, the group with the lowest resulting LCM is chosen and its values added to ap. This procedure is repeated until ap reaches the desired size. We did not perform any rigorous analysis regarding the optimality of this procedure, but empirically it showed significant improvements over naive selections like uniformly sized bins. This approach is not fast, but it can be precomputed, so the speed is not critical.

4.2 Results

To gather a representative dataset, the procedure previously described in Section 4.1 was used with the range of inputs, shown in Table 4.2. All combinations of the inputs were computed for the final dataset. We applied our algorithm to all those configurations 10 times and measured the corresponding execution times.

Number of flows n	10, 50, 100, 200, 300, 400
Leave Component Chance lcc	$25\%, 50\%, 75\%$
Size Range $[sl,su]$	[100,500]
Period Range $[pl,pu]$ in ms	[100,200], [100,400], [10,310]
Number of distinct periods	10, 20, 30

Table 4.2: Flow selection parameter values used for the evaluation

In Figure 4.2, one can see the impact of the lcc, the size of ap, the period bandwidth $(pu - pl)$ and the number of scheduled flows on the median execution time of each configuration. This figure clearly shows that the more distinct periods one chooses, the longer it takes to schedule all flows. The cause of this is that the resulting hyperperiods get larger and larger the more distinct periods are chosen. This problem is exacerbated if the overall range (the period_bandwidth), from which we can choose values for ap, gets smaller. Additionally, the desired size of ap also plays a role in this. One takeaway here is that for narrower period bandwidths the size of ap should also be decreased to keep the resulting hyperperiods small.

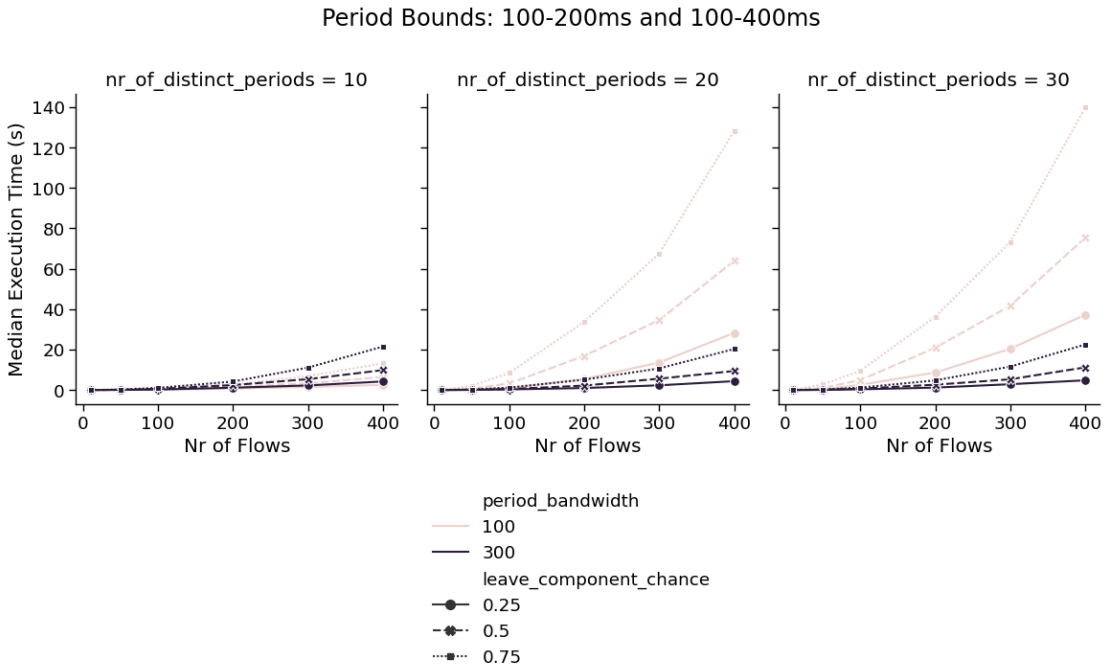

Figure 4.2: Median execution time comparison

Another thing that is apparent is that the more flows travel across bridges, the harder it gets to find a schedule. This is caused by a higher utilization of those bridging links which makes it harder to find valid offsets for all frames and it also increases the hyperperiod on those links because more distinct flows will travel across the same connection.

This effect is also evident in Figure 4.3. There, one can also observe that this is less critical if the same number of individual periods is chosen from a broader range of possible periods. This way, the overall hyperperiod will be lower. Another fact to note here is that the number of periods we choose also plays an important role. We can see again that this is amplified if we only choose the periods from a smaller range because this drives up the hyperperiod.

In conclusion, the composition of individual periods and their distribution plays an important role in the efficiency of the algorithm. This is probably true for all approaches that are based on the concept of considering the hyperperiod of all flows.

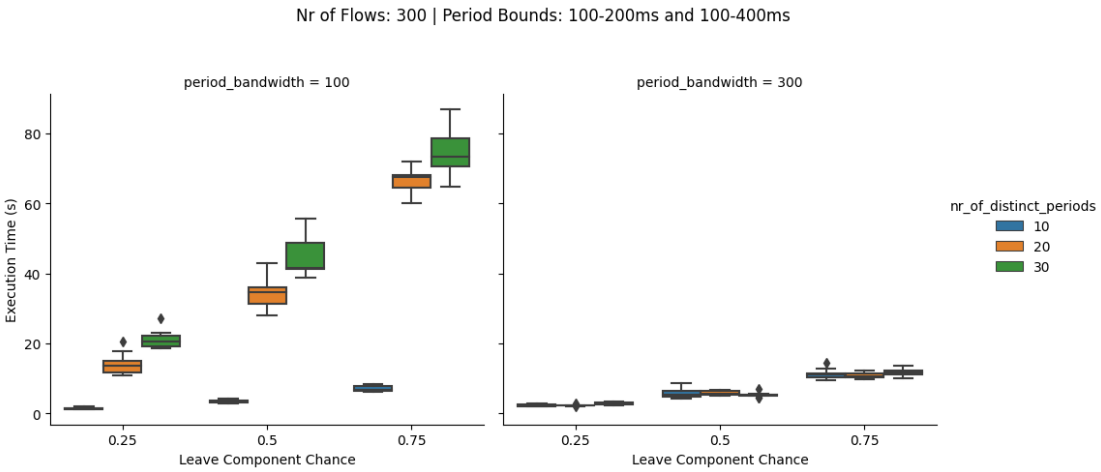

Figure 4.3: Influence of leave component chance (lcc) and number of distinct periods (ndp) on execution time

CHAPTER 5

Conclusion and Future Work

In this paper, we implemented the existing approach to compute TSN Schedules presented by Raagaard et al. ([RP17]). Furthermore, we added additional bookkeeping to the algorithm to enable faster recalculations after topology changes. In the future, a direct comparison between the two implementations might prove useful to see the runtime impact of the bookkeeping efforts. If this implementation fares worse, this gap might be reduced by reimplementing critical parts in C or Rust to speed up the computations like it was done in the original publication. A prime candidate would be the function which is responsible for the 'folding' of a UtilizationList into a shorter interval to determine possible sending offsets.

From our evaluation, we can see that it is beneficial to be mindful of the impact of certain parameters during the initial design of the network and the messages being sent across it. If possible, one should take care to design those properties to make it as easy as possible for the algorithm to successfully find schedules quickly. As this is an NP-complete problem, the potential savings are big. Our suggestions are to select the periods of flows that will travel across the same links in such a way that the resulting hyperperiod on those links stays as small as possible.

Further evaluation needs to be performed on the quality of the solutions after changes to the topology. Another useful and interesting test would be to check for situations where the implementation hits its limits in terms of successfully recalculating schedules. In our tests, we saw that it is very successful when a node just moves within a network component, but the success chances are reduced if a node moves to a different network segment, because this increases the load on the bridges. Thus, it gets harder to schedule flows across those bridges in general. In this regard, some tests to check how the recalculated schedule compares to a full reschedule might be insightful. On a similar note, we would like to compare the different proposed post-processing strategies like ASAP (As-soon-as-possible) or ALAP (As-late-as-possible) and their variations from the

original paper in terms of the chances for successful recalculations. In those tests, we would like to see if it makes any difference how the individual frames are placed into the respective feasible regions.

Another future addition would be to take the proposed GRASP metaheuristic approach from the original authors and improve it with the available bookkeeping to reduce the runtime of the calculations with their proposed delta evaluation steps. This could then be used to improve the schedules after they have been deployed. An approach like this could eventually also 'heal' the schedules that are computed after changes to the network to be more optimal and have higher chances of success in case of further changes. This approach could be similiar to the one described by Pozo in [PP19] for Time-Triggered Ethernet. There, they apply a fast patch operation to keep the network running and afterwards refine the schedule again in a way that the messages are placed such that the success chances for future patch actions are increased. We suspect that this would result in schedules which have a better chance of being able to be adapted to new situations when new changes happen. This combination could then have favorable properties for flexible networks. On the one hand, we have fast recalculations to react to changes, on the other hand, we can improve this imperfect solution over time and bring it closer to an optimal solution with minimal impact and downtimes of the network.

As a practical improvement we foresee the restructuring of the code into a proper library with a clearly defined Application Programming Interface (API) as a valuable step. This will enable easier integration of the algorithm into a larger system which would be concerned with the general management of a TSN network.

List of Figures

List of Tables

Bibliography

[Com17a] Ulgorash (Wikimedia Commons). TSN IEEE 802.1qbv example with frame preemption. https://commons.wikimedia.org/wiki/File:TSN_frame_pre-emption.svg, Apr 2017. Accessed on 2022-07-09.

[Com17b] Ulgorash (Wikimedia Commons). TSN IEEE 802.1qbv example with guard band. https://commons.wikimedia.org/wiki/File:TSN_IEEE_802.1Qbv_Example_with_guard_band.svg, Apr 2017. Accessed on 2022-07-09.

[Com20] Encoder Products Company. Industrial Ethernet communication protocols. https://www.encoder.com/hubfs/white-papers/WP-2019_Industrial-Ethernet-Protocols/wp2019-industrial-ethernet-communication-protocols.pdf, Oct 2020. Accessed on 2022-01-30.

[Dij59] E. W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1(1):269–271, 1959.

[DN16] 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, page 203–212, New York, NY, USA, 2016. Association for Computing Machinery.

[FS02] Max Felser and Thilo Sauter. The fieldbus war: History or short break between battles? pages 73 – 80, 02 2002.

[Gmb21] NetTimeLogic GmbH. TSN Basics. https://www.nettimelogic.com/resources/TSN%20Basics.pdf, Dec 2021. Accessed on 2022-07-03.

[GSDP17] Marina Gutiérrez, Wilfried Steiner, Radu Dobrin, and Sasikumar Punnekkat. Synchronization quality of IEEE 802.1AS in large-scale industrial automation networks. In *2017 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 273–282, 2017.

[HSS08] Aric A. Hagberg, Daniel A. Schult, and Pieter J. Swart. Exploring network structure, dynamics, and function using networkx. In Gaël Varoquaux, Travis Vaught, and Jarrod Millman, editors, *Proceedings of the 7th Python in Science Conference*, pages 11 – 15, Pasadena, CA USA, 2008.

[IET13] IETF. TSN Tutorial. `https://www.ieee802.org/802_tutorials/2013-03/8021-IETF-tutorial-final.pdf`, Mar 2013. Accessed on 2022-07-03.

[LLJ+20] Qing Li, Dong Li, Xi Jin, Qizhao Wang, and Peng Zeng. A simple and efficient time-sensitive networking traffic scheduling method for industrial scenarios. *Electronics*, 9(12), 2020.

[MMR99] R. Macchiaroli, S. Mole, and S. Riemma. Modelling and optimization of industrial manufacturing processes subject to no-wait constraints. *International Journal of Production Research*, 37(11):2585–2607, 1999.

[MP02] Alessandro Mascis and Dario Pacciarelli. Job-shop scheduling with blocking and no-wait constraints. *European Journal of Operational Research*, 143(3):498–517, December 2002.

[NDR16] Naresh Ganesh Nayak, Frank Dürr, and Kurt Rothermel. Time-sensitive software-defined network (TSSDN) for real-time applications. *Proceedings of the 24th International Conference on Real-Time Networks and Systems*, 2016.

[PP19] Francisco Manuel Pozo Pérez. *Methods for Efficient and Adaptive Scheduling of Next-Generation Time-Triggered Networks*. PhD thesis, Mälardalen University, 2019.

[Res] Microsoft Research. Z3 prover. `https://github.com/Z3Prover/z3`. Accessed on 2022-04-03.

[RP17] Michael Lander Raagaard and Paul Pop. Optimization algorithms for the scheduling of IEEE 802.1 time-sensitive networking (TSN). Technical report, DTU Compute Technical University of Denmark, Lyngby, 2017. `http://www2.compute.dtu.dk/~paupo/publications/Raagaard2017aa-Optimization%20algorithms%20for%20th-.pdf`.

[RZCP20] Niklas Reusch, Luxi Zhao, Silviu S. Craciunas, and Paul Pop. Window-based schedule synthesis for industrial IEEE 802.1Qbv TSN networks. In *2020 16th IEEE International Conference on Factory Communication Systems (WFCS)*, pages 1–4, 2020.

[Sei23] Sebastian Seitner. `https://gitlab.auto.tuwien.ac.at/safety/networkscheduling`, 2023. Source code for this work.

[SSN19] 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)*, pages 69–77, 2019.