Institute of Parallel and Distributed Systems

University of Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Masterarbeit

Desynchronization Algorithms for Fast ILP Solutions of TDMA Schedules in IEEE Time-Sensitive Networks

Patrick Schneefuss

Course of Study:	Softwaretechnik
Examiner:	Prof. Dr. Kurt Rothermel
Supervisor:	Dr. Frank Dürr
Commenced:	November 18, 2020
Completed:	May 18, 2021

Abstract

Time-Sensitive Networking (TSN) is extending standard *IEEE 802.3 Ethernet* with deterministic real-time capabilities, which are essential for many networked *real-time systems*. It does so by implementing a *time-division multiple access* (TDMA) scheme.

To operate correctly, this requires the presence of a global schedule to manage access times. However, the methods and algorithms to compute such schedules are not part of the *TSN* standard. The calculation of a schedule for the TDMA scheme of Time-Sensitive Networks is generally $\mathcal{NP}$-hard. A popular method to solve such optimization problems is to formulate them as an *integer linear program* (ILP). State-of-the-art ILP solvers can solve the problem to proven optimality at the expense of very high runtimes. To combat the latter, some ILP solvers provide the user with the possibility to specify an initial set of hints for the variables of the ILP. Existing ILP-based approaches for TSN did not exploit this method so far. Therefore, it is an open research question whether and how much TSN scheduling could be improved by providing hints to the ILP solver.

In this work, we thus propose to split up the solution process. First, we generate a set of hints. Afterwards, we supply the hints to the ILP solver, which is then executed on the original scheduling problem. The essential question in this process is now, how a good heuristic to calculate hints that support the ILP solution looks like. The generated hints should approximate a feasible solution of the scheduling problem; however, their calculation must be fast to speed up the overall process.

To this end, we transfer the primitive of *desynchronization*, which originally had been proposed for wireless ad-hoc networks, to the TSN domain. Intuitively, to avoid collisions of transmitted packets as required by a valid TDMA schedule, we space out the transmission times of network traffic by adapting the sending times.

In this thesis, we present a total of five different desynchronization algorithms for Time-Sensitive Networks. Our results show that by supplying initial desynchronization hints to the ILP solver, the runtime to compute a first feasible solution to the scheduling problem is sped up by at least a factor of six compared to an execution without any provided hints.

Kurzfassung

Time-Sensitive Networking (TSN) erweitert standardisiertes *IEEE 802.3 Ethernet* um deterministische Echtzeit-Fähigkeiten, die essentiell für viele *Echtzeitsysteme* sind. Dies wird durch die Implementierung eines Zeitmultiplexverfahrens (TDMA) erreicht.

Für eine korrekte Funktionalität wird hierfür ein globaler Zeitplan (engl. schedule) benötigt, der die Zugriffszeiten koordiniert. Konkrete Algorithmen und Mechanismen, um einen solchen Zeitplan zu berechnen, sind allerdings nicht Teil des TSN Standards. Die Berechnung eines Zeitplans für TSN Netzwerke ist allgemein $\mathcal{NP}$-schwer. Ein beliebter Weg zur Lösung solcher Optimierungsprobleme ist die Formulierung des Solchen als *ganzzahlig lineares Programm* (ILP). ILP-Lösungsprogramme sind in der Lage, diese Probleme beweisbar optimal zu lösen, allerdings auf Kosten sehr hoher Laufzeiten. Um die Lösung zu beschleunigen, bieten einige ILP-Lösungsprogramme dem Nutzer die Option, eine Reihe an initialen Hinweisen für die Variablen des ILPs anzugeben. Bereits existierende, auf ILP basierende TSN-Ansätze haben dieses Verfahren allerdings bisher nicht ausgenutzt. Deshalb ist es eine offene Forschungsfrage, ob und um wieviel sich *TSN-Scheduling* durch bereitgestellte Hinweise verbessern lässt.

Hierzu wird in dieser Arbeit eine Zweiteilung des Lösungsprozesses vorgeschlagen. Zuerst wird eine Reihe an Hinweisen generiert. Daraufhin werden diese Hinweise an ein ILP-Lösungsprogramm weitergebenen und dieses auf die ursprüngliche Problemstellung angesetzt. Die grundlegende Frage in diesem Prozess ist nun, wie eine geeignete Heuristik zur Berechnung initialer Hinweise zur Unterstützung des ILP-Lösungsprogrammes aussieht. Einerseits sollen die generierten Hinweise eine bestmögliche Annährung an eine geeignete Lösung des Scheduling-Problems sein. Andererseits muss die Berechnung dieser Hinweise schnell erfolgen, sodass der gesamte Lösungsprozess beschleunigt wird.

Zu diesem Zweck wird in dieser Arbeit das Prinzip der *Desynchronisation*, welches ursprünglich für kabellose Ad-hoc-Netzwerke vorgeschlagen wurde, auf das TSN-Scheduling Problem übertragen. Intuitiv gesehen können hiermit die Übertragungszeiten des Netzwerkverkehrs gleichmäßig verteilt werden, sodass Kollisionen von übertragenen Paketen vermieden werden.

In dieser Arbeit werden fünf Desynchronisationsalgorithmen für TSN Netzwerke vorgestellt. Die Evaluierung dieser Algorithmen zeigt, dass die Lösungszeit zur Berechnung einer ersten geeigneten Lösung durch Angeben von initialen Desynchronisations-Hinweisen um mindestens einen Faktor von sechs verkürzt wird.

Contents

List of Figures

List of Algorithms

1 Introduction

With recent efforts towards digitization, many people see the industry amid its fourth revolution. Often referred to as *Industry 4.0* [ZLZ15] or the *Industrial Internet of Things* [BHCW18], the focus lies on extending current manufacturing with new-risen smart technologies and pave the way for completely interconnected, smart factories. An important factor enabling these innovations are so-called *cyber-physical systems*, which connect physical systems such as machines with software components and interconnect them using computer networks. This allows the exchange of information across multiple systems and thus enables to monitor and control the underlying physical machines. Without the need for human interaction, this severely enhances productivity. Since many physical processes, such as the motion control of machines or robots, are safety critical and time-sensitive, it needs to be guaranteed that a system can timely react to events in its environment. This implies the requirements for deterministic latency bounds for data delivery. Systems that are subject to such "hard" deterministic constraints on end-to-end latency are called *hard-realtime* systems.

While there already exist various hard-realtime communication technologies in the aforementioned domains, they currently all rely on proprietary field bus technologies to ensure the satisfaction of their real-time constraints. Having multiple different proprietary solutions in place is not very appealing in the long term, as it hinders interoperability and increases the costs for supporting different technologies. This strongly motivates the usage of standardized networking technologies. While being the prevailing standard for data transmission in computer networks for decades, current Ethernet according to IEEE 802.3 [Ins16a] was designed for data transmission in a local area office setting and is not equipped with the needed level of deterministic realtime capabilities. Therefore, the IEEE *TSN (Time-Sensitive Networking) Task Group* [Ins20] is working on a set of standards to extend standard *IEEE 802.3 Ethernet* with said deterministic realtime-capabilities.

One core mechanism of Time-Sensitive Networking is the so-called *Time-Aware Shaper* (TAS), which controls the order of outgoing frames on the egress ports of a TAS-enabled Ethernet switch by implementing a *time-division multiple access* (TDMA) scheme [Ins16b]. It differentiates the priority of the data frames traversing the network according to a priority value defined in the frame's header. TSN defines eight traffic classes that are mapped to different *queues* for each egress port. Data frames waiting in a queue can only be sent out on the Ethernet link if the queue's corresponding *transmission gate* is currently open. To operate correctly, this requires the presence of a global schedule that manages sending times for time-sensitive traffic and opening or closing events for the corresponding transmission gates. Concrete algorithms and mechanisms to calculate such schedule however are not part of the TSN standard and are to be supplied externally.

Integer Linear Programming (ILP) is one prominent method to model the TAS scheduling problem as constrained optimization problem and solve it by readily available ILP solvers such as Gurobi [Gur21] or CPLEX [Cpl09]. For instance, [DN16] introduces the *No-wait Packet Scheduling Problem* (NW-PSP), which maps the more commonly known No-wait Job Shop Scheduling Problem

[MP02] to the TSN domain. Besides prohibiting an overlap in time of two transmissions over a common network link, the no-wait constraint further enforces a *zero-queueing* policy. Consequently, once a frame is received by some switch of the network, it is to be forwarded immediately without queueing. Its transmission is not to be deferred by any cross traffic. Like the No-wait Job Shop Scheduling Problem, the NW-PSP is proven to be $\mathcal{NP}$-hard [SL86] similar to many other ILP-based scheduling approaches.

One major advantage of an ILP solver is that it can solve the problem to proven optimality. More so, it provides proven optimality bounds to every solution it computes along the way, which cannot be obtained when the scheduling problem is solved by heuristics [PSRH15][HGF+20][DN16] only. Unfortunately, the runtime of an ILP Solver increases drastically when scaling up the problem instance, so that in practise they often become unusable.

To combat this, Gurobi provides the user with the option to supply either so called *hints* or *start values* for the variables of the ILP. If the user is confident that some variable will have a certain value in high quality solutions to the problem, he consequently should supply this information as a hint [Gur21]. While hints can always be supplied to the program, regardless of their quality, start values require the supplied information to already form a feasible solution to the constraints of the corresponding ILP.

The possibility to speed up ILP solutions through hints has not been exploited in the TSN literature so far. Therefore, it is an open research question whether or how much hints for the TAS scheduling problem could speed up ILP solutions. To fill this research gap, we propose a two-phase solution process in this work. In the first phase, we calculate a set of hints to the scheduling problem, which are then fed into the ILP in phase two to support the ILP solver. The basic idea is that by providing initial values that are at least close to a feasible solution of the ILP, the ILP solver can find a first feasible solution faster, as it is guided into a certain direction. By still using the ILP solver in phase two, we can still benefit from ILP solutions with provable optimality bounds.

This directly bring up the following question: What is a good heuristic to calculate hints in the first phase? We require it to both be fast to speed up the overall process, and to be able to approximate feasible solutions of the scheduling problem to support the ILP. To develop such a heuristic for TAS scheduling, we took inspiration from a related TDMA scheduling approach used for wireless ad-hoc networks, called desynchronization [DRN07]. Here, a set of nodes periodically want to perform a task or access a shared resource. They now try to adjust their access times in a way, that they are as far away as possible from the access of any other network node. While desynchronization was originally designed for wireless single-hop sensor networks only, several efforts [DN08][KW09][MK09] have been made to extend the original algorithm [DRN07] to wireless multi-hop topologies. By expanding on these concepts, we now want to transfer desynchronization to the context of Time-Sensitive Networks and the No-wait Packet Scheduling Problem. Here, we are given a set of network flows and try to desynchronize their respective transmission starting times. A flow is a sequence of data frames that is periodically sent from a source host in the network to a destination host.

Intuitively, desynchronizing senders such that their transmissions are far apart in time should reduce the risk of collisions and, therefore, violate the corresponding no-collision constraints of the ILP less frequently. However, in reality it is not straight forward to transfer desynchronization to IEEE Time-Sensitive Networks, as factors like network delays have a heavy impact on these transmission conflicts as well.

In this thesis, we will thus explore how well we can adapt desynchronization to fit our problem at hand and discuss limitations along the way. As main contribution, we will present five different approaches to desynchronize IEEE Time-Sensitive Networks and benchmark each of them thoroughly. Lastly, we explore, how using the results obtained by any of our desynchronization algorithms as hints for the Gurobi ILP solver impacts the solving time and the quality of the obtained solution. Here our results show that the provision of desynchronization hints speeds up the time to calculate a first feasible solution by at least a factor of six.

The structure of this thesis is as follows:

Background: This chapter will provide the reader with a fundamental understanding of all technologies and concepts building the foundations for the contributions of this thesis. It will provide a concise introduction to Time-Sensitive Networking, to the No-wait Packet Scheduling Problem and most importantly to desynchronization.

Related work: Since desynchronization was originally designed for single-hop networks, we will briefly discuss first efforts made to extend the classic desynchronization algorithm to wireless multi-hop topologies.

System Model and Problem Statement: After having introduced first approaches made towards multi-hop desynchronization, we will discuss how the primitive of desynchronization can be transferred to the domain of TSN Networks. We will introduce several assumptions we make going forward, but also argue that some limitations for desynchronization in wireless networks do not apply for Time-Sensitive Networks.

Desynchronization Algorithms for IEEE Time-Sensitive Networks: This chapter forms the main contribution of this thesis. We present five approaches that extend the idea of multi-hop desynchronization presented in previous chapters to the context of IEEE Time-Sensitive Networks. We will further discuss advantages and limitation of each approach along the way.

Evaluation: Finally, we thoroughly benchmark the desynchronization algorithms presented earlier. To do so, we introduce two desynchronization scores to measure the degree of desynchronization. This enables to evaluate if some of the presented algorithms could even be used as a standalone solution to the TSN scheduling problem. Furthermore, we examine how the produced solutions of each desynchronization approach impact the solving process of the Gurobi ILP solver when being supplied as hints.

2 Background

In this chapter we provide a brief overview over the fundamental concepts required to understand the approaches proposed later in this thesis. We first illustrate the *Time-Sensitive Networking* switching model according to [Ins16b]. Especially a clear understanding of the network delays in Time-Sensitive Networks will be essential for the algorithms presented in the following chapters. Secondly, we discuss the *No-Wait Packet Scheduling Problem* [DN16], which models the scheduling problem for Time-Sensitive Networks considered in this work. Lastly, we introduce the primitive of *desynchronization* [DRN07], originally designed for wireless single-hop networks, which we later expand to the domain of Time-Sensitive Networks to generate a set of hints as input for the corresponding No-wait Packet Scheduling Problem.

2.1 Time-Sensitive Networking

Time-Sensitive Networking (TSN) is a set of standards defined by the IEEE TSN Task Group [Ins20] that extend standard IEEE 802.3 networks [Ins16a] with deterministic real-time capabilities by implementing a *time division multiple access* (TDMA) scheme. One core principle of Time-sensitive Networking is the *Time Aware Shaper* (TAS) [Ins16b].

A TAS-enabled network switch, depicted in Figure 2.1, is divided into an ingress pipeline and an egress pipeline. Once a data frame arrives at the switch, it enters the ingress pipeline. Here, the incoming frame is buffered until the last bit has been completely received. The process of storing an incoming frame in a buffer until complete reception is called *store-and-forward* switching.

After a frame is completely buffered the checksums are verified and the switching logic determines the correct egress port according to the destination address. The frame is then relayed to the determined egress port. The time needed for this process is called the *processing delay* of a network switch.

Each egress port of a TAS-enabled network switch is equipped with up to eight *queues*. The frames to be forwarded are distributed amongst the queues of an egress port according to the *3-bit PCP value* of the Ethernet header. Using the PCP value, a TAS-enabled switch thus enables a *stream-based scheduling* by distinguishing between high-priority traffic and low-priority traffic, also simply called best-effort traffic in the following.

To enable the prioritization of traffic classes, each queue of an egress port is controlled by its *transmission gate*, which can be either “open” or “closed”. A frame may only be dequeued and transmitted if it is the first frame in its queue and the queue’s transmission gate is currently open. The state of all transmission gates of an egress port is defined by a *schedule*, called the *gate control list*. The gate control list contains multiple entries, where each specifies a state for all gates of an

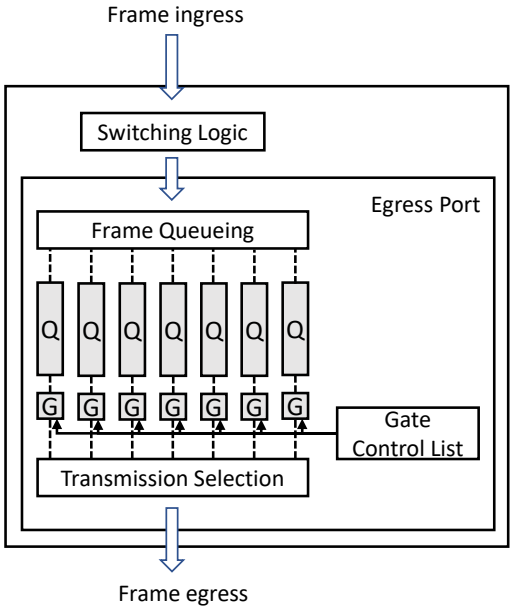


Figure 2.1: Switching model of a TAS-enabled network switch

egress port. Further, every entry in the gate control list defines a timespan for which the respective state holds true. If multiple gates of an egress port are opened at once, the transmission selection selects the frame to be transmitted next based on a selection algorithm such as *strict priority*.

The *transmission delay* refers to the time it takes to modulate every bit of a data frame onto the transmission medium and depends on the link speed as well as the frame length [DK+14]. With a fast link speed, more bits can be transmitted over the medium per second, and thus the transmission delay will be smaller than for links with a slow speed. Once a bit is modulated onto the transmission medium, it needs to traverse this medium towards the receiving interface. This timeframe is called the *propagation delay*, which is influenced by the propagation speed of the medium (e.g., *copper* or *fibre*) and the traversed distance [DK+14]. As a result, the time between transmission start at a sending interface and complete reception of the frame at the receiving interface is defined by the sum of the transmission delay and the propagation delay. These delays are further visualized in Figure 2.2.

After transmitting a frame over an Ethernet link there must be a gap equal to the transmission time of at least 96 bits [Ins16a]. This idle period is called the *inter frame gap* (ifg). It is necessary to enable the receiver to successfully distinguish between two consecutive incoming frames. With a link speed of 100 Mbps, the ifg amounts to exactly 960 nanoseconds. This implies that after starting transmission of some frame over a network link, this link becomes available again after both transmission delay and inter frame gap have passed.

The time a frame spends in a queue of an egress port is called the *queueing delay* [DK+14]. The longer a frame is queued, the higher is its end-to-end latency. While the previously listed delays are inevitable, the queuing delay can be minimized. If a centralized network controller knows the traffic patterns and paths of high-priority traffic in advance, gates of every switch port along its path can be opened at the correct times to enable minimal end-to-end delays. During this time,

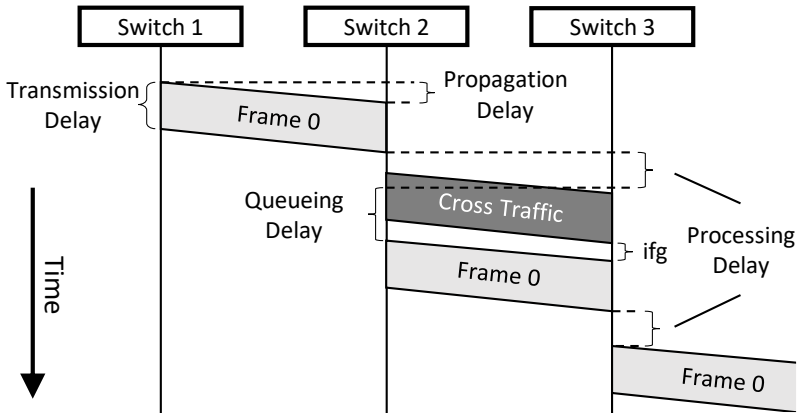


Figure 2.2: Network delays a frame experiences in Time-Sensitive Networks

especially the gates of best-effort queues are to be closed to avoid an interference with best-effort transmissions, which may introduce queueing delay for high-priority traffic. Ideally, the gate control lists are configured with such precision, that *zero queueing* is achieved for high-priority traffic. Here, an incoming frame will be forwarded immediately, as upon arrival the corresponding queue is empty and the transmission gate open.

A complete schedule for a Time-Sensitive Network is thus comprised of a gate control list per egress port, and the starting times for all sent traffic. For the purpose of this thesis, we consider all traffic scheduled to be high-priority traffic. We thus consider a schedule to only feature starting times for high-priority flows and opening and closing operations, as well as their corresponding timestamps, for high-priority queues.

2.2 No-wait Packet Scheduling Problem

As concrete mechanisms to calculate a schedule for the TDMA scheme of Time-Sensitive Networks are not included in the TSN standard, [DN16] proposes the *No-wait Packet Scheduling Problem* (NW-PSP) to model the scheduling in such networks. The No-wait Packet Scheduling Problem reduces to the more commonly known *No-wait Job Shop Scheduling Problem* (JSP) [MP02] and is thus $\mathcal{NP}$-hard [SL86].

In the classic *Job Shop Scheduling Problem* there are a set of jobs $\mathcal{J}$ that must be executed on a set of machines $\mathcal{M}$. A job j can be further divided into a set of operations O_j, which must be executed in a specific order. A job is completed, once all its operations have been executed. For each operation, there is only one dedicated machine capable of executing it. Picture an assembly line in a beer brewery. Different operations may be to fill up a bottle or to seal the bottle with a bottle cap. There is a dedicated machine for each of these tasks, however the bottle can only be sealed with a cap after it has been filled.

Once a certain operation of some job has been started on a machine, it is not to be interrupted until completion. Thus, we cannot stop an operation after it has been started, to resume it at a later point in time, possibly on a different machine. Additionally, no two operations of the same job are to be

executed on the same machine, and no two operations of a job can be executed in parallel. This implies that the number of machines needed to complete a job is equal to the number of operations making up the job.

The goal of the JSP is now to assign a starting time and a machine to every operation $o \in O_j$ of all jobs $j \in \mathcal{J}$, such that the following criteria are always satisfied:

- At starting time t of operation $o \in O_j$, all previous operations o' of job j have been completed.
- From the starting time t of operation o, which is being executed on machine m, up until completion of this operation, no other operation o' is scheduled to be executed on the same machine m.

The optimal result to this problem also minimizes the makespan. The makespan is defined as the time that has passed between starting the first job and completing the last job.

The *No-wait Job Shop Scheduling Problem* (NW-JSP) extends this definition by further prohibiting any downtime between two consecutive operations of a job. Once an operation has been completed on some machine, the consecutive operation must immediately be started on another machine. Consequently, if a machine m is idle at some point in time, and there exists some operation o that can be executed on this machine, we cannot take advantage of the idle machine, if after completion of o there is no idle machine available to immediately process the consecutive operation o'.

The time to complete a job is thus defined by the sum of the execution times of its operations only. Furthermore, we see that it is now redundant to assign a starting time t to every operation o, as the starting time of the first operation of a job is enough to create a full schedule. The starting times of consecutive operations are well defined due to the *no-wait* constraint.

In Time-Sensitive Networks, instead of a set of jobs that must be processed, we have a set of time-sensitive flows $\mathcal{F} = \{F_1, F_2, \ldots, F_n\}$ that must be delivered. Here a flow is a sequence of packets sent between a certain source host and a destination host. Instead of machines, we have a set of network switches. However, [DN16] emphasizes the necessity to make a distinction between machines in the classic NW-JSP and network switches in the NW-PSP. Each network switch can forward incoming traffic over several different outgoing ports, depending on the location of the destination host of the corresponding flow. A network switch can forward multiple flows in parallel, if they are being forwarded over different outgoing ports. These parallel transmission further do not conflict with each other. Thus, the set of machines of the NW-JSP is not mapped to the network switches, but rather to the outgoing ports of the all network switches. All ports of the network are denoted by the set $\mathcal{P} = \{P_1, P_2, \ldots, P_3\}$ [DN16].

Each flow has a predefined path that defines an ordered set of network ports over which it must be transmitted to reach the destination host. Consequently, network ports perform forwarding operations on incoming flows. As a result, the delivery of a flow F_i can be split up into several forwarding operations $O_i = \{O_{i,1}, O_{i,2}, \ldots, O_{i,n}\}$ that must be executed in a defined order by a defined set of network ports [DN16].

The *no-wait* constraint further implies, that a flow cannot be queued at any switch in the network. Once a flow is received, it must be forwarded as soon as the processing by the respective switch has finished. As already mentioned for the classic NW-JSP, this simplifies a schedule for the

TDMA mechanism in Time-Sensitive Networks to just starting times of the high-priority flows. The respective gate operations and the time at which they are to be performed can be deduced via the different network delays introduced in the previous section.

Further, [DN16] defines the cumulative network delay $\mathcal{D}_{i,j}$ up to and including the successful execution of the forwarding operation $O_{i,j}$ of flow F_i as:

$$\mathcal{D}_{i,j} = (j-1)(d^{prop} + d^{proc}) + \sum_{k=1,\dots,j} d^{trans}_{i,k}$$

Note that this cumulative network delay does not include the propagation delay and processing delay corresponding to the last forwarding operation. Thus, after $\mathcal{D}_{i,j}$ the transmission operation $O_{i,j}$ has been completed, but the transmission has not yet been received and processed by the consecutive network switch.

We extend the ILP formulation of the NW-PSP presented in [DN16] by including the inter frame gap introduced in the previous section. The resulting integer linear program is given as:

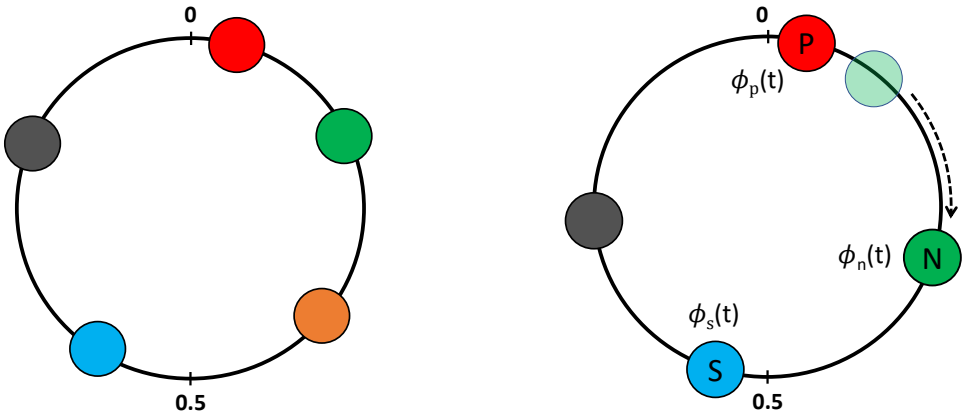
$$\begin{aligned} &\min C_{max} \\ &\text{subject to} \end{aligned} \tag{2.1}$$

$$\begin{aligned} &\forall \{O_{i,k}, O_{j,l}\} \in K : \\ &t_i - t_j + D_{i,k} - D_{j,l-1} - d^{prop} - d^{proc} + d^{ifg} \leq c x_{i,k,j,l} \end{aligned} \tag{2.2}$$

$$\begin{aligned} &\forall \{O_{i,k}, O_{j,l}\} \in K : \\ &t_j - t_i + D_{j,l} - D_{i,k-1} - d^{prop} - d^{proc} + d^{ifg} \leq c(1 - x_{i,k,j,l}) \end{aligned} \tag{2.3}$$

$$C_{max} \leq C \tag{2.4}$$

Here C_{max} denotes the time at which all flows have been received by their corresponding destination host. Minimizing C_{max} thus minimizes the makespan of the schedule. t_i and t_j in Constraints 2.2 and 2.3 denote the time, at which the flows F_i and F_j start being transmitted by their source host. K denotes a set of forwarding operations that are to be executed on the same network port and thus may conflict with each other. The Constraints 2.2 and 2.3 themselves then ensure that the execution of no two conflicting operations overlap in time. More precisely, they require a transmission operation $O_{i,k}$ to be concluded and the inter frame gap to have passed, before the conflicting operation $O_{j,l}$ is ready to be executed, or vice versa. As only one of the two constraints must and can be fulfilled, [DN16] introduces the binary variable $x_{i,k,j,l} \in \{0, 1\}$ to convert the disjunction into a conjunction as required by integer linear programs. Further, c is defined as a very large constant that will never be smaller than the left side of the inequality of either Constraint 2.2 or 2.3 (*Big M method*). Consequently, depending on the value of $x_{i,k,j,l}$, exactly one of the two constraints is "active" as its right-hand side becomes zero. The remaining constraint will then always evaluate to true. If an initial upper bound for the length of the schedule is provided, Constraint 2.4 further restricts the makespan to be smaller than this given upper bound. Solutions that satisfy all constraints of the integer linear program are called a feasible solution. Feasibility itself does not require the makespan to be optimal. If to an integer linear program no solution satisfying all constraints can be derived, then the problem instance is considered infeasible.



(a) Phases visualized on the desynchronization ring

(b) Phase update of a node on the desynchronization ring

Figure 2.3: The desynchronization ring

2.3 Desynchronization

The primitive of synchronization often is described by the phenomenon of a swarm of fireflies pulsating in unison. Here, the fireflies continuously adjust their own pulse according to their neighbourhood, until eventually the pulses of all fireflies are synchronized [DRN07]. In contrary, desynchronization, first introduced in [DRN07], aims to achieve the logical opposite. In the context of computer networks, a node does not try to synchronize the execution of a task with other network nodes, but rather tries to perform the task as far away in time from other nodes as possible. Desynchronization thus can be used to manage access to a shared resource, or more importantly for this thesis, to organize a collision-free TDMA schedule for wireless sensor networks. Desynchronization was originally designed for single-hop wireless sensor networks, where all nodes can directly communicate with each other.

[DRN07] bases the desynchronization algorithm on the pulse-coupled oscillator framework presented in [MS90]. In this framework, there are n nodes that each want to perform a periodic task. The length of this period T, also called cycle, is uniform between all nodes. Thus, the nodes can be modelled as oscillators with frequency $1/T$. [MS90] further introduces the notion of phases. The phase $\phi_n(t) \in [0, 1]$ of a node n at time t, where 0 and 1 are identical phases, denotes the current progression of this node towards completing its period. While the length of the period T is uniform between all nodes, their progression towards completion is not. The phase of a node increases steadily over time by a fixed rate. The phases of the individual nodes are thus often visualized as beads moving forward in clockwise direction on a ring (cf. Figure 2.3a).

Once a node n reaches $\phi_n(t) = 1$, it fires a message to inform other nodes about the completion of its cycle and resets its phase to 0. In the desynchronization algorithm presented in [DRN07], the firing of a node corresponds to the broadcast of a wireless message. All nodes listen for firing messages of their peers. Based on the knowledge gained through firing messages of peers, a node may update its phase to improve the current desynchronization, additionally to continuously increasing its phase over time. From the perspective of a node n, the node that fired last before itself is considered

the *predecessor* of n, and the node that fires first after itself is considered the *successor* of node n [DRN07]. Node n now waits until it received a firing message from both its predecessor and successor.

Both predecessor and successor are called the phase neighbours of node n [DRN07]. Node n has no global knowledge about the current phase of any other node in the network. However, the phases of its neighbours can be approximated by n by determining the respective phase difference between itself and its predecessor, as well as between itself and its successor. Calculating the phase difference is straight forward from the standpoint of node n, as the phases of both phase neighbours are known to be 1 upon firing. When node n picks up the firing message of its predecessor, the phase difference thus corresponds exactly to the difference between its own phase $\phi_n(t)$ and 1. The same applies for the phase difference to node n's successor. Consequently, the calculation of phase differences does not require any global knowledge. Immediately upon reception of the firing message of its successor, node n will now update its phase to eventually realize a desynchronized state of the network. For this, at time t of the update, node n moves its phase to the mid-point of the approximated values $\phi_p(t)$ and $\phi_s(t)$ (cf. Figure 2.3b). The new phase of node n after the update at time t is thus defined as

$$\phi_n(t) = (\phi_p(t) + \phi_s(t)) \div 2$$

if it holds that $\phi_s(t) > \phi_p(t)$ [DRN07]. However, as the desynchronization can be represented by a ring where the phases 0 and 1 are "glued" together [DRN07], it can also hold that $\phi_p(t) \geq \phi_s(t)$. In this case, the new phase $\phi_n(t)$ is given as

$$\phi_n(t) = ((\phi_s(t) + 1 + \phi_p(t)) \div 2) \quad mod \quad 1$$

When updating its phase, node n assumes that the phase difference to both its phase neighbours has not changed in the time passed between approximation of their phases and updating its own phase. Consequently, it is also possible that node n bases its update on stale data, as either its predecessor or successor meanwhile might already have jumped to some other point on the ring.

The authors additionally provide formal prove that regardless of the initial state of all phases, the system will eventually converge into a desynchronized state. In such desynchronized state, the phase difference between two neighbouring phases on the desynchronization ring will always approach $\frac{1}{N}$ for networks consisting of N nodes [DRN07].

3 Related Work

In this chapter, we will introduce two different approaches extending the previously described primitive of desynchronization to wireless multi-hop topologies. On the one hand, [DN08] explores the behaviour of the classic desynchronization algorithm discussed in Section 2.3 on these more complex topologies. It further discusses practical limitations introduced by the *hidden terminal problem*. M-DESYNC [KW09] on the other hand moves away from the classic desynchronization approach of [DRN07] and proposes a new desynchronization algorithm for acyclic wireless multi-hop networks that is resilient to the hidden terminal problem.

3.1 Desynchronization for Multi-Hop Topologies

In [DN08], Degesys et al. followed up on their first publication on desynchronization for wireless single-hop sensor networks [DRN07] by discussing extensions for multi-hop topologies. When using desynchronization to determine a TDMA schedule for single-hop wireless sensor networks, we have seen in the previous section that the sending phases for nodes of the network are being distributed evenly over a given period, so that a node's sending time resides as far away in time as possible from the sending times of other nodes.

As in single-hop wireless networks the wireless medium can only be used by one sender at a time, this ordering of sending times - or in the context of TDMA schedules, this spacing of TDMA slots - yields a high probability that two transmissions of separate network nodes do not collide. Obviously, for a network consisting of many nodes and a relatively short cycle, collision-freeness cannot be guaranteed completely, as the respective interval between sending times of two nodes might be smaller than the time needed to fully transmit a message. For multi-hop wireless sensor networks, nodes can send messages in parallel if they are located far enough apart from each other in the network, so that their transmissions can never interfere. Consequently, to achieve a desynchronized state in a wireless multi-hop network, we do not need to actively space out sending times of two network nodes, whose transmissions will never interfere due to their distance. [DN08] thus describes multi-hop desynchronization as closely related to the graph colouring problem [JT11]. Here, given a graph $G(V, E)$, the task is to assign a colour to every vertex $v \in V$, so that no two neighbouring vertices of G have the same colour. The set of used colours is to be kept minimal.

Translating multi-hop desynchronization to the graph colouring problem, we are given the graph $G(V, E)$ where V is the number of nodes in the network. Two nodes v and u are connected by an edge $e \in E$, if the nodes are in transmission range of each other. The nodes in transmission range of some node n are called n's one-hop neighbours. The graph $G(V, E)$ thus corresponds to the communication graph of the network [DN08]. Having found a solution to the graph colouring problem, a phase can be assigned to every colour. By then desynchronizing these phases it is ensured that the sending times of nodes with possible transmission conflicts are desynchronized.

On the other hand, phases of nodes without any transmission conflicts may also be equal. As a result, there are less distinct phases considered in the desynchronization process, thus leading to larger spaces between the sending times of two nodes within each other's transmission range. In the context of TDMA schemes, this also implies that the slots assigned to each node are larger. For wireless networks, only considering neighbouring network nodes and their phases during desynchronization is already implicitly achieved by [DRN07], as only the firing messages of network peers in transmission range will be picked up. However, given the properties of wireless networks, we can still see an obvious problem with the described approach due to the so-called *hidden terminal problem*.

The hidden terminal problem occurs when two wireless nodes, say A and C, both transmit a message to a third node, say B. Here, A and C are not in transmission range of each other, B however is in transmission range of both nodes A and C. If both, node A and node C, transmit a message to node B, neither A nor C can detect the transmission of the other party, although their messages will interfere at the receiver (B). With the above-described idea on desynchronizing wireless multi-hop networks, we may end up with A and C being assigned the same colour, and consequently the same phase. Therefore, in the graph G not only an edge between a node n and all its one-hop neighbours is introduced, but also an edge between node n and all its two-hop neighbours. The set of two-hop neighbours of node n is defined as the union of the one-hop neighbours of all of node n's one-hop neighbours. This graph G is then called the constraint graph or conflict graph of the network. To address the hidden terminal problem, the desynchronization process can then be executed on the constraint graph of the network instead. As two-hop neighbours of a node n in the communication graph will be direct neighbours of n in G, n will also be able to pick up on their firing messages. For desynchronization on the constraint graph, [DN08] presents results in comparison to execution on the communication graph. It is to be noted however that desynchronization on the constraint graph is a purely theoretical approach for wireless sensor networks, as a node n is considering phases of its two-hop neighbours when updating its phase. The phase of its two-hop neighbours, however, is not known by n, as they are not in transmission range of n and the firing messages can consequently not be received in practise [DN08].

3.2 M-DESYNC

Given the described practical limitations introduced by the hidden terminal problem, [KW09] picks up on the idea of desynchronization for wireless sensor networks and proposes M-DESYNC, a localized multi-hop desynchronization algorithm. Resilient to the hidden terminal problem, M-DESYNC addresses the link scheduling problem for single hop wireless networks, as well as acyclic multi-hop wireless networks. It aims to assign a slot within the communication period to each node, during which no other possibly interfering node may transmit a message simultaneously.

It defines the degree of a node as the number of its one-hop neighbours in the communication graph and further introduces the local max degree D. The local max degree D_n of a node n is the maximum of its own degree and the degrees of its one-hop neighbours [KW09]. A node n now divides its period T into $|t_n|$ different slots of size $\frac{T}{|t_n|}$ [KW09]. This also implies that the number of slots and their sizes may differ for two nodes of the network. We further see how setting t_n to the number of n's two-hop neighbours would solve the hidden terminal problem, but wastes

bandwidth, as some two-hop neighbours may be able to transmit simultaneously without their messages interfering. [KW08] proves that the number of needed slots $|t_n|$ for a node n is at least $D_n + 1$.

A node obtains its local max degree by exchanging degree information with its one-hop neighbours before the desynchronization process begins [KW09]. This information exchange must be repeated on every change to the topology, such as the joining of a new node. At the start of the communication period, a node now chooses one of its time slots as transmission slot. If the respective slot has been reached, it probes the transmission medium for conflicting transmissions. If a transmission by some neighbouring node is detected, it aborts its own transmission, probes the remaining slots, and tries again in the next period. If no conflicting transmission is detected, the node broadcasts a starting message in the chosen slot to all its one-hop neighbours [KW09]. In all slots but the single slot chosen for transmission, a node listens for starting messages of its one-hop neighbours. If such a starting message is received, the node relays this message to its own set of one-hop neighbours. By doing so, it is ensured that starting messages of any node will always reach the set of its two-hop neighbours. If now in some slot a node b receives starting messages from two of its one-hop neighbours a and c, where a and c are not within each other's transmission range, the relayed messages of node b will inform a and c about the conflict. In case of such conflict, each affected node will back off with some probability, eventually resolving the conflict. After backing off, a node picks a new transmission slot and tries transmitting again in the next period [KW09].

To reduce the number of collisions introduced by this random selection of initial slots, and thus to speed up the desynchronization process, M-DESYNC [KW09] further proposes two strategies for optimized slot selection. The first approach is called *modulo pre-coloring*. Here, all nodes of the network are assumed to have a unique identifier. Each node can now calculate the initial time slot chosen for transmission with the modulo operation of its identifier and the number of its total transmission slots. This strategy works purely on local information and thus does not introduce any overhead in form of additional message exchanges. The second strategy introduces a priority-based back off mechanism. Here, M-DESYNC assigns each node of the network a priority value. This priority information will be included in starting messages of a node, as well as in the relayed starting messages. In case of a conflict, the node with the smaller priority will always back off and pick a new slot for the next period, while the higher priority node keeps the current slot. This decreases the time to convergence, as at no time both conflicting nodes back off. Their results show that the benefit of modulo pre-colouring is most noticeable in dense networks with high average node degree, while the priority-based back off mechanism accelerates convergence regardless of the network density [KW09].

We see that both presented approaches are designed for decentralized networks and assume local knowledge only. In the following sections we thus discuss how we can transfer their concepts to a Time-Sensitive Network with centralized control that has a global view onto the entire network. Moreover, these related approaches do not explicitly consider the network delay between nodes in the multi-hop network, which is well known in a Time-Sensitive Network with global network control.

4 System Model and Problem Statement

After having discussed the basic concepts needed to understand the work of this thesis, we will now introduce our system model and the assumptions we make, while also reiterating on the problem we are trying to address.

4.1 System Model

Initially, we are given a set of network *flows*. A flow is a sequence of packets that is sent from a source host in the network to a destination host. The flow is defined by an identification number and a given flow path. This flow path includes the source host, the set of network switches forwarding the flow in the order the flow traverses them, and the destination host. The path length of a flow is defined by the number of network links it is transmitted over until it reaches the destination host. We further call the transmission of a flow over a link that leads up to the nth network switch (or host) on the flow path the nth hop of this flow. In modern networks, network links are commonly full duplex. This means, that a link connecting two parties of the network is bidirectional. More so, it enables both connected parties to transmit data over this link simultaneously. Thus, even if there exists only one link between two switches S and T of a network, we later treat this as two separate unidirectional links. As a result, we will treat network links and outgoing network ports as equivalent, as each outgoing port is connected to exactly one network link. Likewise, each network link is connecting exactly one outgoing port to an incoming port.

We say two flows are in *conflict* if both their flow paths share at least one common network link. While there may be multiple shared links between two flows, we often refer to the first shared link as the *conflict link* of the two flows.

The packets of a flow are generally sent periodically, where the time between two transmissions is referred to as the *cycle* of the flow. Here, all flows are considered to be high-priority flows with stringent requirements on end-to-end latency.

We assume a stream-based scheduling and prioritization of traffic as defined by the *Time-Aware Shaper* [Ins16b] described in Section 2.1. Ultimately, we now want to obtain a *schedule* that assigns a starting time to each flow and features timestamps for necessary gate operations of the transmission gates of TAS-enabled network switches, so that we can guarantee the successful delivery of each flow within a certain upper bound on latency. The starting time of a flow is defined as the time where the source host of this flow modulates the first bit onto the transmission medium. We assume that the schedule is computed by a centralized network controller with a global view onto the entire network.

After all flows have been delivered and all gate operations defined by the schedule have been executed, the schedule is repeated. We also refer to the length of this schedule as its *cycle*. For simplicity of our model, we assume that the cycle of all flows in our network is equal to the cycle of our schedule. Consequently, each flow will be sent out exactly once per cycle of the schedule.

To model the scheduling process in IEEE Time-Sensitive Networks, we use the *No-wait Packet Scheduling Problem* [DN16] described in Chapter 2. As it prohibits the queueing of a network flow at any switch along its path, the schedule is well defined by only the starting times of the network flows. Given a set of starting times that form a feasible solution to the No-wait Packet Scheduling Problem, it is then simple to compute the times at which the gates of high-priority queues must be opened and closed by tracing the flow on its path through the network. The concrete time a flow is received by some switch along its path can be calculated by using the various network delays described in Section 2.1.

Here, we assume that the frames of all flows sent across the network are of the same size and that all network links operate with the same link speed. This already implies that the transmission delay for all flows is equal as well. We furthermore assume all network switches to have equal processing delay, and all network links to have the same propagation delay. Due to the *no-wait* constraint of the No-wait Packet Scheduling Problem we further only consider solutions as feasible, where no flow is queued at any point in the network. At every network switch, any incoming flow is to be forwarded immediately after it has been processed. As a result, the time for two distinct flows to traverse n links in the network will always be equal, as all factors impacting the total transmission time are as well. If two conflicting flows arrive at a network switch at the same time and are to be forwarded over the same outgoing port, we say that the flows collide with each other. Technically however, one of the flows would be forwarded immediately, while the second flow would be stored in a queue until the medium is available for transmission again. As we do not allow for queueing, we will thus refer to this behaviour as a collision. More precisely, a collision not only occurs if the two flows arrive at the exact same time, but also if one of the flows arrives shortly after the other but is processed by the switch before the corresponding egress port is available again.

As we aim to produce hints for the No-wait Packet Scheduling Problem, which is based on the Job-Shop Scheduling problem, we also prohibit that a flow is sent at the end of one cycle, but due to its transmission time received by the destination host only in the following cycle. This is called a wrap around. Lastly, we assume, that all flows that are to be scheduled are routed over a shortest path from the source host to the destination host.

Solutions to the scheduling problem can be obtained by formulating this problem as an integer linear program and feeding it to an ILP Solver. While many different scheduling approaches use similar ILP definitions, we focus on the integer linear program described in Section 2.2

4.2 Problem Statement

While being capable of solving the No-wait Packet Scheduling Problem to optimality and providing optimality bounds for every computed solution, the runtime of an ILP solver usually is very high. To speed up the time it takes to find a feasible solution to the problem, state-of-the-art ILP solver provide an option to define initial values for the variables of the integer linear program. These values

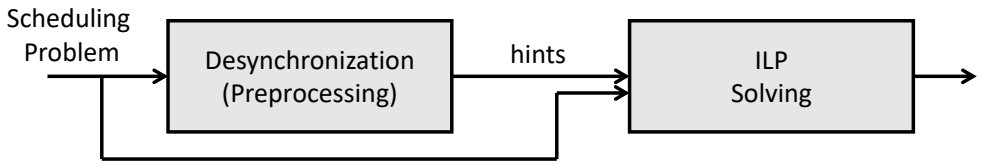


Figure 4.1: Two-phase solution process

ideally form a feasible solution to the scheduling problem already. However, using so-called *hints* also allows to assign an initial set of values for the ILP variables, which violates some constraints of the corresponding integer linear program.

Thus, we propose a two-split of the solving process (cf. Figure 4.1). In the first phase, we want to quickly generate a set of hints for the corresponding NW-PSP ILP with the properties defined in the system model by using a fast heuristic. The hints themselves are not required to form a feasible solution to the NW-PSP. In the second phase, we solve the original scheduling problem using an ILP solver, however we initialize the variables of the integer linear problem with the set of hints generated in the first phase. By doing so, we aim to noticeably reduce the solving time of the ILP solver. As various heuristics [PSRH15][HGF+20][DN16] can compute some arbitrary, non-optimal solution to the NW-PSP in comparably little time, we especially want to speed up the time it takes the ILP solver to calculate a first feasible solution to the problem. As a result, we can derive a fast first feasible solution, but also obtain provable upper bounds on its optimality.

Consequently, we want to provide hints that approximate a feasible solution to the scheduling problem as closely as possible. Intuitively, by spacing out the starting times of the flows, we reduce the risk of collisions between any pair of flows. Less collisions imply less violations of constraints of our corresponding No-wait Packet Scheduling Problem and thus a higher quality solution to provide to an ILP solver as a set of hints. To this end, we will adapt the primitive of desynchronization [DRN07], as described in Section 2.3, to IEEE Time-Sensitive Networks and our concrete problem at hand.

Desynchronization was designed to enable the derivation of a collision-free TDMA schedule for wireless single-hop networks by spacing out the transmission times of all nodes in the network. Multiple approaches [DN08][KW09][MK09] have then expanded desynchronization to wireless multi-hop topologies. As already described in Chapter 3, not all nodes in a multi-hop topology need to desynchronize their sending time with each other. For multi-hop desynchronization, it is sufficient if a node only considers all nodes in its transmission range, as well as its two-hop neighbours in the network topology to address the hidden terminal problem. Here, every network node is assigned an individual sending time, or rather a sending phase. For the remainder of this thesis, we will adopt the notion of phases introduced in [MS90] and described in Chapter 2. If a node is receiving incoming traffic which is to be forwarded further, it does not immediately do so. Rather, the node queues incoming traffic after reception until its respective sending phase has been reached, and only then it will forward the frame.

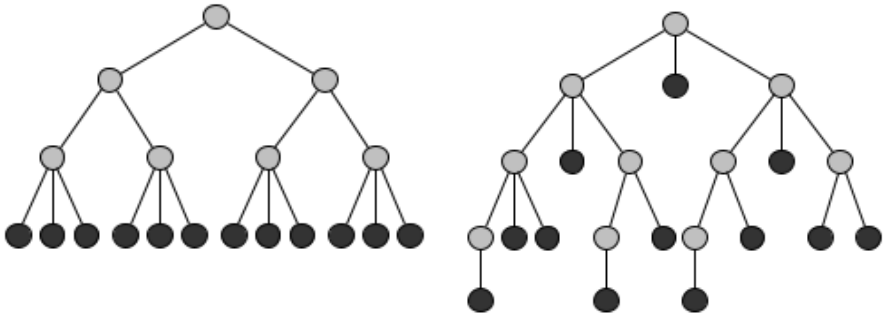
It is clear to see that this approach is not suitable for the purpose of this thesis. We require the calculation of a schedule without any queueing delay. If a network switch receives an incoming frame, this frame must be forwarded immediately after it has been processed by the switch. A desynchronization of the network nodes' sending times thus is no viable option for desynchronization in IEEE Time-Sensitive Networks, as we strongly require a node to be able to forward multiple flows at different points in time throughout the cycle.

Instead, as previously motivated, we can desynchronize the starting times of all flows that are to be scheduled. However, if two flows f_1 and f_2 do not conflict on any link of the network and thus cannot collide with each other, it is not necessary to consider f_2 when desynchronizing f_1 and vice versa. As a result, the graph that is to be desynchronized does not correspond to the communication graph of the network, but rather models the conflict relations between flows. Here, two flows are only connected by an edge if they conflict with each other. As two flows either traverse at least one common network link and thus may collide, or else are completely independent of each other, we further see how in this conflict graph only the neighbours of a flow must be considered when desynchronizing it. We especially emphasize that neighbours of flows in the conflict graph are not to be confused with neighbours of switches in the network topology. By neighbours of a node n in the network topology, we refer to network nodes that are either in transmission range of n in a wireless network or connected to n by a cable in wired networks. By neighbours of a flow f in the conflict graph, we refer to a flow that shares at least one common network link with f and thus conflicts with f.

Furthermore, when desynchronizing a wireless network, a node n informs all other nodes about its current phase by firing a broadcast message upon completion of its cycle. These messages enabled other network nodes to approximate the phase of node n. Any updates to the phase of a node n are thus not immediately known by other nodes, as they first need to propagate through the network. As a result, it is possible that a node bases its own phase update on stale data, because updates from other relevant nodes have not yet been detected. With self-organization in mind, this enables the algorithm to adapt to a dynamic network topology, where nodes may leave and enter the network at any given time.

However, as we are trying to derive a schedule for the TAS mechanism of Time-Sensitive Networks, we are assuming a static topology. The calculation of the schedule is an external process and is to be carried out before any data is sent. Consequently, as we assume the presence of a centralized network controller with global view, the updated phase of a network flow is immediately visible to all other network flows and a flow can access the current phases of all other network flow at any time. This renders the exchange of messages obsolete, as they are not needed to determine the phases of conflicting flows before an update anymore. Instead, we update the phases of all network flows iteratively in a round robin fashion. The global view further implies that we do not need to increase the phases by a certain rate over time as described in Section 2.3. This steady increase over time only guaranteed that broadcast messages are exchanged in regular intervals. Thus, unless a flow actively updates its phase through an update, the phase will stay constant. As a result, we will refer to a phase of some flow f by just ϕ_f instead of $\phi_f(t)$.

To briefly summarize the problem again, using a fast heuristic we want to generate a set of hints for the NW-PSP ILP that specifies initial values for the starting times of network flows. While these hints are not required to form a feasible solution to the scheduling problem, they should at least closely approximate a feasible solution. By doing so, we aim at speeding up the solving process of the utilized ILP solver. To this end, we adapt desynchronization to the domain of IEEE



(a) Balanced tree topology **(b)** Imbalanced tree topology

Figure 4.2: Considered topologies for desynchronization of Time-Sensitive Networks. Host nodes are coloured black.

Time-Sensitive Networks. By utilizing desynchronization to space out the flows starting times, we aim at avoiding as many collisions as possible. As a result, the produced desynchronization should also violate only very little constraints of the corresponding NW-PSP ILP. As opposed to desynchronization in wireless networks, we further assume the presence of a centralized network controller, which has a global view onto the network and the desynchronization process.

The topologies considered when designing desynchronization algorithms for Time-Sensitive Networks and used for benchmarking in the later evaluation are different variations of tree topologies. These topologies are very relevant for Ethernet in general, as a spanning tree is embedded in any network topology. Here we will distinguish between balanced (Figure 4.2a) and imbalanced tree topologies (Figure 4.2b). Leaf nodes of the tree represent the set of network hosts, while all inner nodes represent network switches. While in the balanced tree topologies all leaf nodes must reside on the same level of the tree, this restriction is relaxed for imbalanced tree topologies. As a result, there is only a very small set of different path lengths between two network hosts in balanced trees. Imbalanced trees introduce a wider variety of path lengths between two network hosts, which should also impact the desynchronization algorithms, as we will outline further in the following chapter.

5 Desynchronization Algorithms for IEEE Time-Sensitive Networks

After having described our system model and having defined the general desynchronization problem to be solved, we will now present five desynchronization algorithms for IEEE Time-Sensitive Networks. Most of the algorithms are based on the multi-hop desynchronization approach of [DN08], however one algorithm also draws inspiration from [KW09]. We further discuss advantages and limitations of all algorithms. While the algorithms do not aim to yield a feasible solution to the No-wait Packet Scheduling Problem, they try to compute a fast approximation of how a feasible solution might look like.

5.1 Naive Desynchronization

The first desynchronization approach for IEEE TSN networks is a simple adaptation of the desynchronization algorithm for multi-hop wireless networks presented in [DN08]. As it does not account for any network properties such as the delays during the desynchronization process, we will refer to it as Naive Desynchronization for the remainder of this thesis.

The algorithm is based on a constraint graph G. The set of flows to be scheduled form the set of vertices of G. Two vertices of the graph are connected by an edge if the two corresponding flows conflict with each other, i.e., they share at least one common network link on their flow paths. Thus, in the following we will also refer to this graph as the conflict graph.

As mentioned in the previous chapter, the cycles of our flows are all equal and thus we set the cycle of the schedule to match the flow cycles. During desynchronization, every flow f will have a phase ϕ_f in the $[0, 1)$ interval. After the desynchronization algorithm terminates, the resulting phases of all flows will then be mapped to the concrete sending time in nanoseconds. The larger the phase ϕ_f is, the higher the sending time will be.

This sending time not only has to be smaller than the cycle of the schedule, but it must also be small enough, so that the flow can be completely transmitted and received by the destination host before the current cycle ends. This corresponds to the constraint of the No-wait Packet Scheduling Problem which prohibits a *wrap-around* of flows. Consequently, simply mapping from the $[0, 1)$ interval for desynchronization to the cycle time is not sufficient, as it does not enforce the mentioned constraint. Thus, before beginning with the desynchronization process, we first calculate the latest possible sending time L for which we can guarantee that all flows sent before this time limit is exceeded will reach their destination host on time. The total time a flow takes from transmission start at the sending host to reception on the destination host can be calculated using the introduced network delays. This process is depicted in Algorithm 5.1 (cf. line 10). The time a flow needs to completely traverse one link of its flow path is defined by the sum of the transmission delay and the propagation

Algorithm 5.1 Latest Starting Time

```
1:  function LatestStart(cycle)
2:      F ← getAllFlows()
3:      L ← cycle
4:      for f ∈ F do
5:          t_latest ← cycle − transmissionTime(f)
6:          if t_latest < L then
7:              L ← t_latest
8:      return L
9:
10: function transmissionTime(f)
11:     ρ ← len(flowPath(f)) −1
12:     T ← ρ ∗ (T + PP) + (ρ − 1) ∗ PC          // T, PP, PC are the network delays
13:     return T
```

delay of the network. Naturally, the number of links traversed is one less than the number of hosts and switches on its flow path. After the flow is received by some switch, the total transmission time is further increased by the processing delay of this switch. Only at the receiving host the flow does not need to be processed upon reception. Once the total transmission time is calculated for each flow, we use the longest transmission time and subtract it from the given cycle length, as shown in Algorithm 5.1 (cf. line 1). This yields the desired latest starting time L. This approach might not optimally use the whole cycle, as very short flows could possibly be sent at a later point in time and still be delivered before the cycle ends. However, as we do not aim to produce an optimal solution with our desynchronization, but rather try to approximate a feasible solution to scheduling problem, we argue that this rather conservative approach suffices for the purpose of this thesis.

Having calculated the latest starting time L, we end up with an interval of possible sending times between 0 and L. We can now establish a mapping between the $[0, 1)$ phase interval for desynchronization and the calculated $[0, L)$ interval. We can convert from one interval to the other by either dividing by L or multiplying by L, depending on the start and target interval. We then randomly initialize the phases of all flows with a uniform distribution across the $[0, 1)$ interval and begin the desynchronization.

The full Naive Desynchronization algorithm can be seen in Algorithm 5.2. It tries to iteratively update the phases of all flows until a desynchronized state is achieved. In each iteration, the phase of every flow is updated exactly once. When trying to determine a new phase for a flow f we will consider the phases of all flows which are neighbours of f in the conflict graph G (cf. Algorithm 5.2, line 9). We will denote this set of flows as C_f. All other flows do not need to desynchronize their phases with f, as a collision can be ruled out and the flows may even be transmitted simultaneously. Intuitively, one can picture each flow having its own desynchronization ring, where only its own phase and all phases of flows in C_f reside on. If the phase of some flow f is updated, this update will also be reflected on the rings of all flows that f conflicts with.

Analogous to [DRN07], we define the phase distance $\Delta(\phi_f, \phi_g)$ between two flows f and g as the difference of their two phases ϕ_f and ϕ_g. To determine the phase distance $\Delta(\phi_f, \phi_g)$, we measure the size of the interval between the two phases, starting from the phase of flow f, going in clockwise direction. This also implies, that $\Delta(\phi_f, \phi_g) = 1 - \Delta(\phi_g, \phi_f)$. We further define the absolute phase

Algorithm 5.2 Naive Desynchronization

```
1:  function NaiveDesync(max_iterations, threshold)
2:      F ← getAllFlows()
3:      cg ← createConflictGraph()
4:      initializeRandomPhases(F)
5:      maxShift ← 1
6:      n_iteration ← 1
7:      while maxShift > threshold and n_iteration < max_iterations do
8:          for all f ∈ F do
9:              C_f ← getConflictGraphNeighbors(cg, f)
10:             phaseShift ← updatePhase(f, C_f)
11:             if phaseShift < maxShift then
12:                 maxShift ← phaseShift
13:         n_iteration ++
14:
15: function UpdatePhase(f, C_f)
16:     s ← determineSuccessor(f, C_f)
17:     p ← determinePredecessor(f, C_f)
18:     if φ_s > φ_p then
19:         φ_new ← (φ_s + φ_p)/2
20:     else
21:         φ_new ← (φ_s + φ_p + 1)/2
22:         if φ_new ≥ 1 then
23:             φ_new ← φ_new − 1
24:     φ_old ← φ_f
25:     φ_f ← φ_new
26:     return |Δ(φ_new, φ_old)|
```

distance $|\Delta(\phi_f, \phi_g)|$ as the minimum of $\Delta(\phi_f, \phi_g)$ and $\Delta(\phi_g, \phi_f)$. Consequently, it also holds that $|\Delta(\phi_f, \phi_g)| = |\Delta(\phi_g, \phi_f)|$. Intuitively, the absolute phase distance $|\Delta(\phi_f, \phi_g)|$ corresponds to the actual distance of the two phases on the desynchronization ring, as we can either move in clockwise or counter-clockwise direction.

Of all flows in C_f we now determine the flow p with minimum $\Delta(\phi_p, \phi_f)$ and the flow s with minimum $\Delta(\phi_f, \phi_s)$ (cf. Algorithm 5.2, line 16). We call s the direct successor of f, which is the flow with the next highest phase on f's ring, and we call p the direct predecessor of f, which is the flow with the next lowest phase on f's ring. Here we also account for a wrap around in the $[0, 1)$ interval. This means, if we do not find any flow in C_f with a phase smaller than ϕ_f, we instead pick the flow in C_f with the largest phase as the predecessor of f. Likewise, if there is no flow in C_f with a larger phase than ϕ_f, we pick the flow in C_f with the smallest phase as the successor of f (cf. Figure 5.1). If a flow f does not conflict with any other flow and thus is the only flow on its desynchronization ring, we cannot and must not determine a successor and predecessor, as the phase of f does not need to be desynchronized and may remain at the value assigned through random initialization. If flow f only conflicts with exactly one flow g, then this flow g will act as both the successor and predecessor of f. To eventually achieve a desynchronized state, we now move ϕ_f as far away from both ϕ_p and ϕ_s as possible. As a result, the new phase of f will then be

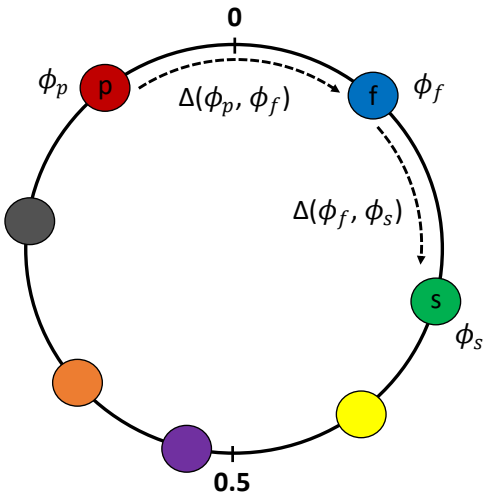


Figure 5.1: Successor and predecessor on the desynchronization ring

the mid-point between ϕ_p and ϕ_s (cf. Algorithm 5.2, line 18). To correctly calculate the mid-point, it is important to check whether the phase of s is larger than the phase of p, in which case we can simply calculate the new phase of f as the average of ϕ_p and ϕ_s. However, if the phase of s is not larger than the phase of p due to the above-mentioned wrap around, then we need to adapt the calculation of the mid-point accordingly. The mid-point is then calculated by

$$((1 + \phi_s) - \phi_p) \div 2 \quad mod \quad 1$$

We repeat this procedure for every flow of the network. This constitutes one iteration.

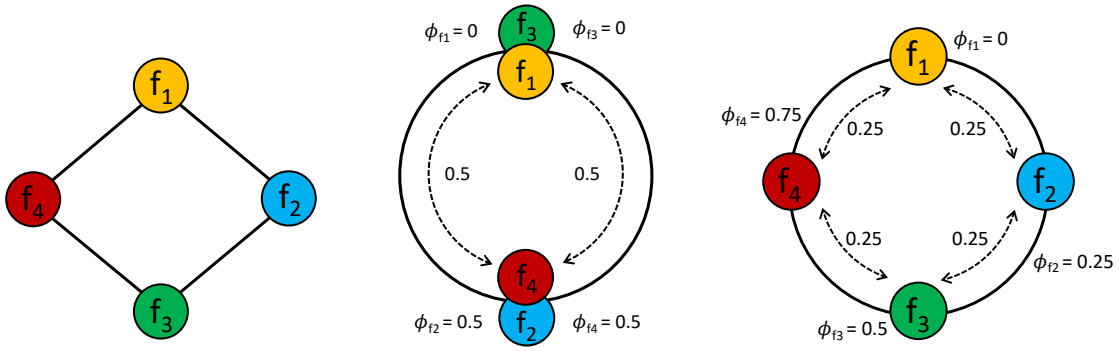
We will execute as many iterations as needed until any termination criterion is met (cf. Algorithm 5.2, line 7). The criteria for termination are the following:

Convergence: During one iteration, the phase of no flow f has been shifted by more than some predefined threshold. The shift corresponds to the absolute phase difference $|\Delta|$ between the old and new phases of flow f, as a shift by 0.9 can be interpreted as only a shift of 0.1 in counter-clockwise direction. If the absolute shift is lower than the defined threshold, we say the desynchronization has converged.

Iteration limit: The number of executed iterations now exceeds a predefined iteration limit. This also implies that within the defined maximum iteration limit the desynchronization process has not converged by fulfilling the first termination criterion.

While desynchronization in wireless single-hop networks was proven to converge eventually in [DRN07], [DN08] has made efforts to extend this prove to wireless multi-hop networks. On simple topologies their results suggested that the desynchronization for multi-hop networks converges as well. However, no formal prove on the convergence of the desynchronization was supplied for these topologies. As this approach is an adaptation of the wireless multi-hop desynchronization presented in [DN08], we thus cannot give any guarantees on the convergence of this approach either and will further examine the behaviour regarding convergence experimentally in the later evaluation.

Here we emphasize that convergence does not necessarily imply a perfect desynchronization, where for each flow the absolute phase distance to its predecessor and successor is maximized. Assume we



(a) Conflict graph of four flows **(b)** Global optimum **(c)** Local optimum

Figure 5.2: Convergence at local and global optimum

are given four flows f_1, f_2, f_3 and f_4. Let their conflict graph be a circle, thus every flow conflicts with exactly two other flows (cf. Figure 5.2a). To this end, let f_1 and f_3, as well as f_2 and f_4 be the pairwise independent flows. We see how $\phi_{f_1} = \phi_{f_3} = 0$ and $\phi_{f_2} = \phi_{f_4} = 0.5$ gives a perfect desynchronization, where the absolute phase distance between any two conflicting flows is 0.5 (cf. Figure 5.2b). While the absolute phase distance between f_1 and f_3, as well as between f_2 and f_4 is zero, these flows do not conflict with each other and this consequently does not affect the desynchronization. In this state, the phase of every flow is located exactly at the mid-point of its predecessor and successor, leading to convergence. However, now assume a desynchronization state with $\phi_{f_1} = 0$, $\phi_{f_2} = 0.25$, $\phi_{f_3} = 0.5$ and $\phi_{f_4} = 0.75$ (cf. Figure 5.2c). The absolute phase distance between any two conflicting flows now only amounts to 0.25. Still, the phase of every flow is exactly at the mid-point of both its predecessor and successor, leading to convergence in an overall slightly worse desynchronized state. We follow that the Naive Desynchronization algorithm can converge either at a local optimum or a global optimum.

After the algorithm has terminated by either converging or reaching the upper limit of iterations, we can then determine the concrete timestamp at which the flow f has to be sent within the schedules cycle by converting its final phase $\phi_f \in [0, 1)$ back to the $[0, \mathrm{L})$ interval calculated in advance.

We have previously mentioned that the phases of all network flows are initialized uniformly at random before we begin with desynchronization. Consequently, the order in which the flows are initially scheduled is randomly determined as well. When we are updating the phase of a flow f, we determine the new phase based on the phases of the flow's predecessor and successor. For the election of predecessor and successor, all flows conflicting with f are taken into consideration. The successor of f is the closest flow among all the conflict flows in clockwise direction on the ring, and f's predecessor is the closest flow in counter-clockwise direction on the ring. As the update now moves f towards the mid-point of its predecessor and successor, we see that with that update the total order of the flows does not change. Consequently, after updating f, neither does the predecessor of f change, nor does the successor. This poses the question, whether the initial order defined by the random phase initialization is in fact unchangeable during the Naive Desynchronization algorithm. However, different flows may have different sets of conflicting flows that are taken into consideration when determining predecessor and successor. Figure 5.3 gives a simple example, where we see the respective desynchronization ring of two flows f (green) and g (red) with different sets of conflicting flows. More precisely, we see that the set of flows conflicting with g is a subset of the set of flows conflicting with f, only missing the brown flow compared to f.

When now updating the phase of g, f is found as predecessor and the orange flow as successor. When moving g to the midpoint of these phases, we see that from a global perspective g now overtakes the brown flow. Consequently, we observe that the order of flows may change over the course of the desynchronization. This is however limited to flows that do not conflict with each other. If two flows conflict with each other, one phase cannot overtake the other and their order on the ring remains fixed. In a worst-case scenario, where all network flows conflict with each other, every flow considers the same set of flows when updating its phase. As a result, the initial order will persist throughout the complete desynchronization process.

Furthermore, the question remains how meaningful the desynchronization of starting times is for Time-Sensitive Networks. Assume we are given two conflicting flows f and g with a conflict link l. Further assume that the phases of the two flows are perfectly desynchronized and thus there is a time gap t between their starting times. We now want to know to which degree the size of the gap t influences the likelihood of a collisions between the two flows. Unfortunately, the conflict link l might be reached by both flows on different network hops. For flow f link l might be the second link it traverses, while for flow g it is the fourth link along its flow path. Consequently, flow g has already traversed more links than flow f when both reach the conflict link. As a result, the total transmission time until flow g reaches the conflict link is longer than the total transmission time of flow f.

Say now flow g is scheduled to be sent earlier than flow f. Even if there is a rather large gap t between the starting times of both flows, we see how we cannot rule out a collision, as the difference in total transmission time up to the conflict link may cancel out the respective time gap. On the flip side, if flow g was to be scheduled after flow f, we can see how the actual difference between their respective arrival times at the conflict link will be even larger than the difference t between their starting times. Theoretically, this is especially pronounced on more complex topologies such as an imbalanced tree, because the number of different path lengths between host nodes is higher than in a balanced tree. As a result, two conflicting flows will have different transmission times up to their conflict link more often.

While this behaviour already poses a problem for Naive Desynchronization on its own, we observe further implications when referring back to the previously discussed order of flows. The example above suggests that even if we conduct two separate runs of Naive Desynchronization, where the distribution of the resulting phases is equivalent, we may end up with solutions of different quality, depending on the final order of flows. This can become problematic for scenarios where the initial ordering does not change over the course of the desynchronization, especially if the phases are initialized poorly. While this can be circumvented easily by executing the algorithm multiple times with different initial configurations, we rather try to address the problem at its roots in the following section by including network delays into the desynchronization process.

5.2 Relative Desynchronization

The weakness of the Naive Desynchronization algorithm is very apparent and has been briefly described in the previous section. The only distinction Naive Desynchronization makes is between conflicting flows and non-conflicting flows. It is however completely oblivious of our network delays. It desynchronizes only the starting times of conflicting flows.

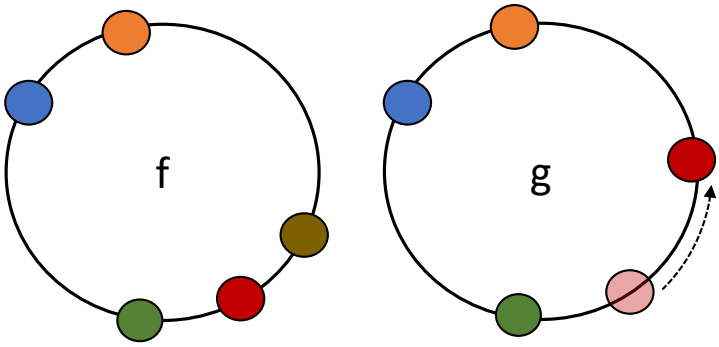


Figure 5.3: Variable ordering of phases during Naive Desynchronization

Given two conflicting flows with an arbitrary difference in starting times, the question is, how strongly the magnitude of this difference impacts the likelihood of a collision between the flows. Intuitively, the larger the distance between both starting times, the lower this likelihood should be. We have seen that this assumption is generally incorrect for Time-Sensitive Networks. Instead, the deciding factor when determining if two flows collide on a conflict link l is the difference between the two points in time where the flows start being transmitted over link l. The time at which a flow starts being transmitted over some link of its flow path is determined by its starting time and the accumulated network delay of this flow up to the respective link. The accumulated network delay is the sum of total queueing delay, transmission delay, processing delay and propagation delay the flow has experienced up until being modulated onto link l. We recall that in our system model we assumed the frames sent by all flows to be of the same size and that all links operate on the same link speed, which leads to uniform transmission delays. We further assumed uniform propagation delay for all network links and uniform processing delay for all network switches. Queueing delay is assumed to be zero throughout the entire network. We conclude that the time between starting transmission of a flow at some switch of the network and reception of this flow at the consecutive switch along its flow path is also uniform between all flows.

We call the time between transmitting the first bit of a flow f over some link l and the flow being completely processed by the next switch the *per hop delay* $\mathcal{PHD}$ of the network with

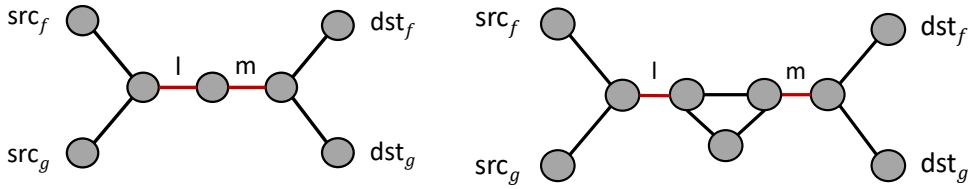
$$\mathcal{PHD} = \mathcal{T} + \mathcal{PP} + \mathcal{PC}$$

where $\mathcal{T}$ is the transmission delay, $\mathcal{PP}$ the propagation delay and $\mathcal{PC}$ the processing delay. The accumulated network delay of a flow up to some link in the network thus is a multiple of $\mathcal{PHD}$. Consequently, the only remaining factor to influence the accumulated network delay up to some link l is the number of hops it takes a flow to reach this link from its source host. We can compute the time at which a flow f is scheduled to start being transmitted over some link l by

$$\mathcal{S}_f + (\eta_{f,l} * \mathcal{PHD})$$

where $\eta_{f,l}$ denotes the number of links f already traversed before reaching link l, and $\mathcal{S}_f$ denotes the starting time of flow f at its source host. Consequently, we can also calculate the time difference between transmission start of two conflicting flows f and g on their conflict link l by

$$|(\mathcal{S}_f + (\eta_{f,l} * \mathcal{PHD})) - (\mathcal{S}_g + (\eta_{g,l} * \mathcal{PHD}))|$$



(a) Tree topology

(b) No tree topology

Figure 5.4: The difference in transmission times of two flows stays consistent over all their conflict links.

For tree topologies in general, but also under our assumption that all flows are routed over shortest paths between their source and destination host only, we can further guarantee that this difference calculated at a conflict link l will remain consistent over all further links m the two flows f and g conflict on. Figure 5.4 further illustrates this. Say flow f and flow g conflict on a link l and a link m, where link l is the first link they conflict on and link m the second link they conflict on. Here, both conflict links are highlighted in red.

If the calculated difference varies for link l and link m, then it would hold that

$$\eta_{f,l} - \eta_{g,l} \neq \eta_{f,m} - \eta_{g,m} \implies \eta_{f,m} - \eta_{f,l} \neq \eta_{g,m} - \eta_{g,l}$$

and thus, the path between link l and link m would be longer for one of the two flows. We see that this is not possible for tree topologies (cf. Figure 5.4a) in general, as no two distinct paths connecting two different links can exist. Also, for other topologies (cf. Figure 5.4b) it would violate the assumption of shortest path routing.

To consider the per hop delay $\mathcal{PHD}$ in the desynchronization, instead of desynchronizing the starting times of flows it is more meaningful to instead desynchronize the transmission start at the conflict link l in some way. For this, we can map the per hop delay $\mathcal{PHD}$ to our $[0, 1)$ interval used for desynchronization as described in the previous section. We will refer to the per hop delay in phase representation as ϕ_{phd}.

Figure 5.5 illustrates an example where two flows conflict on some link l. For flow f, link l is the second link on its flow path and for flow g, link l is the fourth link on its flow path. The current phases of the flows are $\phi_f = 0.5$ for flow f and $\phi_g = 0$ for flow g. We can now calculate the phase $\phi_{f,l}$ that indicates when flow f starts being transmitted over link l by

$$\phi_{f,l} = \phi_f + \phi_{phd} * \eta_{f,l} \mod 1$$

Analogously, we can derive the phase $\phi_{g,l}$, which indicates when flow g starts being transmitted over link l. Assume now ϕ_{phd} to be 0.1 for this example. As flow f traverses three links before reaching link l, it holds that $\phi_{f,l} = \phi_f + 3\phi_{phd} \mod 1$. Flow g only traverses one link before reaching link l and consequently it holds that $\phi_{g,l} = \phi_f + 1\phi_{phd} \mod 1$.

Even though the absolute phase distance $|\Delta(\phi_f, \phi_g)|$ between ϕ_f and ϕ_f is 0.5, the absolute phase distance at the conflict link $|\Delta(\phi_{f,l}, \phi_{g,l})|$ between $\phi_{f,l}$ and $\phi_{g,l}$ is only 0.3 in our example. $\phi_f = 0.5$ and $\phi_g = 0.8$ would consequently be a better desynchronization, resulting in an absolute

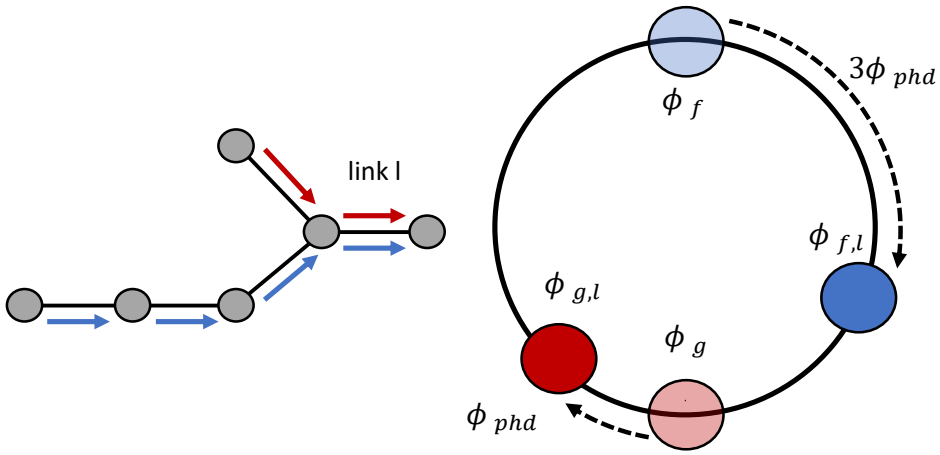


Figure 5.5: Influence of network delays on the desynchronization process

phase distance $|\Delta(\phi_{f,l}, \phi_{g,l})|$ of exactly 0.5 at the conflict link l. As we have reasoned before, the absolute phase distance $|\Delta(\phi_{f,l}, \phi_{g,l})|$ will remain consistent between the two flows f and g on any further conflict link m.

To better reflect the phase difference at the conflict link, we introduce the relative phase $\varphi_{f,g}$ of a flow f from the point of view of a second flow g. For this flow g, we can only compute relative phases of flows conflicting with it.

Definition 5.2.1 (Relative phase). $\varphi_{f,g} = \phi_f - ((\eta_{f,l} - \eta_{g,l}) * \phi_{phd}) \mod 1$

Here, l is some conflict link of flow f and flow g. The phase difference between ϕ_g and $\varphi_{f,g}$ then corresponds to the actual phase difference $\Delta(\phi_{f,l}, \phi_{g,l})$ at the conflict link l since

$$\begin{aligned} &\varphi_{f,g} - \phi_g \\ &= \phi_f - (\eta_{g,l} - \eta_{f,l}) * \phi_{phd} - \phi_g \\ &= (\phi_f + \eta_{f,l} * \phi_{phd}) - (\phi_g + \eta_{g,l} * \phi_{phd}) \\ &= \phi_{f,l} - \phi_{g,l} \end{aligned}$$

Again, assume two conflicting flows f and g with conflict link l, where $\eta_{f,l} = 3$ and $\eta_{g,l} = 1$. Further assume that f is scheduled to be sent before g and thus $\phi_f < \phi_g$. We see that the relative phase $\varphi_{f,g}$ of f from the perspective of g is moved closer to ϕ_g compared to phase ϕ_f. More precisely, we see that $\Delta(\varphi_{f,g}, \phi_g) < \Delta(\phi_f, \phi_g)$, because the relative phase accounts for the longer transmission time of flow f up until the conflict link. $\Delta(\varphi_{f,g}, \phi_g)$ thus represents the actual difference in transmission times of flow f and g over link l. Vice versa, assume now flow g is scheduled to be sent before flow f. We see that the relative phase $\varphi_{f,g}$ of f from the perspective of g is moved farther away from ϕ_g compared to ϕ_f, and thus $\Delta(\varphi_{f,g}, \phi_g) > \Delta(\phi_f, \phi_g)$.

Moving back to improving the original Naive Desynchronization algorithm, we now augment the desynchronization process by incorporating relative phases. We thus refer to the improved algorithm as Relative Desynchronization. The modified parts of the algorithm can be seen in

Algorithm 5.3 Adaptation for Relative Desynchronization

```
1:  function UpdatePhase(f, C_f)
2:      s ← determineRelativeSuccessor(f, C_f)
3:      p ← determineRelativePredecessor(f, C_f)
4:      if φ_{s,f} > φ_{p,f} then
5:          φ_new ← (φ_{s,f} + φ_{p,f})/2
6:      else
7:          φ_new ← (φ_{s,f} + φ_{p,f} + 1)/2
8:          if φ_new ≥ 1 then
9:              φ_new ← φ_new − 1
10:     φ_old ← φ_f
11:     φ_f ← φ_new
12:     return |Δ(φ_new, φ_old)|
```

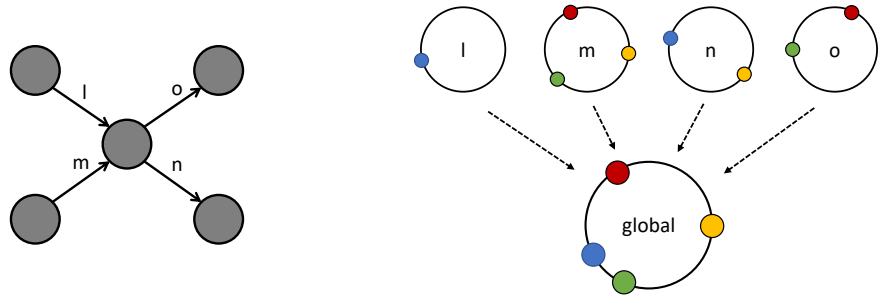
Algorithm 5.3. At its core, the algorithm remains mostly unchanged. For its broad structure, please refer to Algorithm 5.2. What we are adapting however, is the procedure to find the successor s and predecessor p of some flow f when trying to update its phase ϕ_f (cf. Algorithm 5.3, line 2-3). Instead of considering phases ϕ_g of all conflicting flows $g \in C_f$ when updating the phase of some flow f as done in Naive Desynchronization, we consider the relative phases $\varphi_{g,f}$ of all conflicting flows g from the perspective of f. The phases $\varphi_{g,f}$ can be computed for each conflicting flow g after determining the respective conflict link of flow f and flow g. The desynchronization ring of f now consequently contains its own phase ϕ_f and the relative phase $\varphi_{g,f}$ of every conflicting flow g, but not the actual starting phase ϕ_g anymore.

When determining predecessor p and successor s of f, we do not choose the conflicting flows with minimum $\Delta(\phi_p, \phi_f)$ and $\Delta(\phi_f, \phi_s)$, but instead the conflicting flows with minimum $\Delta(\varphi_{p,f}, \phi_f)$ and minimum $\Delta(\phi_f, \varphi_{s,f})$, respectively. When updating the phase of f, we further do not move ϕ_f to the mid-point of ϕ_p and ϕ_s, but rather to the mid-point of $\varphi_{p,f}$ and $\varphi_{s,f}$ (cf. Algorithm 5.3, line 5,7).

We see that by using relative phases during phase updates, we incorporate the network delays into the desynchronization process. As a result, the initial order after random phase assignments has less impact on the desynchronization result. Relative Desynchronization thus addresses both disadvantages mentioned in the previous section on Naive Desynchronization.

5.3 Link Desynchronization

In the previous section on our Relative Desynchronization algorithm, we used the starting phase ϕ_f of a flow f and the per hop delay of the network in phase representation ϕ_{phd} to compute the actual phase, at which the flow is to be transmitted over a conflict link l. The idea behind the desynchronization approach presented in this section is now to not only consider one phase per flow and desynchronize based on these phases, but instead take every transmission event across every link of the network into consideration. By doing so, we hope to enable a more fine-grained desynchronization.



(a) Network topology

(b) Separate desynchronizations per link

Figure 5.6: Visualization of the Link Desynchronization process. Separate desynchronizations are executed for each network link. Their states are then merged into a global desynchronization state.

Given ϕ_f and ϕ_{phd}, we are able to compute the sending phase of a flow f for any of the network links it traverses, not only the conflict links. The phase of flow f when starting transmission on a link l is defined as

$$\phi_{f,l} = \phi_f + \phi_{phd} * \eta_{f,l} \mod 1$$

where $\eta_{f,l}$ is the number of links traversed by a flow f before reaching link l, as defined in the previous section. Thus, if a flow traverses n distinct links on its path to the respective destination host, we can assign n different phases to this flow to keep track of its exact sending phases at any point through the network. The difference between two phases of flow f on two consecutive links on its flow path will always be exactly ϕ_{phd}, as we assume no queueing delay. If link l is the first link on the flow path of some flow f, then the notations ϕ_f and $\phi_{f,l}$ are equivalent. If a link l of the network is now traversed by a set of n flows $\mathcal{F}_l = \{f_1, f_2, \ldots, f_n\}$, we can compute n phases

$$\phi_{f,l} = \phi_f + \eta_{f,l} * \phi_{phd} \mod 1 \qquad \forall f \in \mathcal{F}_l$$

This enables us to execute a separate desynchronization for every link of the network, using the respective phases calculated with the formula introduced above. We will thus call this approach Link Desynchronization. As stated in our system model, we consider a full duplex link in a network topology as two separate unidirectional links. Thus, for one full duplex link we execute two separate desynchronizations. Here, a desynchronization of a link l considers all phases $\phi_{f,l}$ of flows f traversing link l. After the desynchronization of all links has terminated, we then need to merge their states into a single, global desynchronization state to obtain a unique starting time for each flow. This global state then only contains the actual starting phases ϕ_f for each flow f of the network, but not any phases of consecutive network hops, because only the former are relevant for a schedule. Figure 5.6 visualizes the high-level idea of the described algorithm. For simplicity of the graphic, the links displayed in Figure 5.6a are considered to be unidirectional links only. We see that instead of one desynchronization ring per flow as in previous approaches, we can now picture one desynchronization ring per link.

Assume we are desynchronizing a link l traversed by the set of flows $\mathcal{F}_l = \{f_1, f_2, \ldots, f_n\}$. The update of any phase $\phi_{f_i,l}$ of the flows on this link is always based on the same set of flows $\mathcal{F}_l$. This strongly reminds of the classic desynchronization algorithm for single-hop networks, where

the conflict set remained consistent between all network nodes. However, as we are executing separate desynchronizations for each link of the network, we notice that the phase ϕ_f of some flow f is not only influenced by a single desynchronization, but by as many desynchronizations as the number of links f traverses on its flow path. If we now update the phase $\phi_{f,l}$ of some flow f while desynchronizing link l, we must keep in mind that we do not allow for any queueing delay in our network. Consequently, the challenge is to keep the phase distance of size ϕ_{phd} consistent between phases of consecutive hops of this flow f with

$$\phi_{f,l+1} = \phi_{f,l} + \phi_{phd} \mod 1$$

Thus, if we update $\phi_{f,l}$ by shifting it by a distance of m during desynchronization of some link l, we consequently must also apply the same shift to the phases of f on all other links $\mathcal{L}_f \setminus l$ it traverses to stay consistent with our no queueing policy (cf. Algorithm 5.4, line 29). Furthermore, without keeping this consistence between the desynchronizations of different links we will later not be able merge the desynchronized states of every link into a single, global desynchronization state. Since the update of a phase $\phi_{f,l}$ also influences the phases of flow f on all other links it traverses without considering the respective desynchronization state of those links, it should be clear that the initial order of flows can and will change throughout the desynchronization process.

At first glance, the optimization goal of Link Desynchronization seems to be very closely related to Relative Desynchronization, as in both approaches we try to maximize the phase distances between conflicting flows at the actual conflict links. However, in the Relative Desynchronization algorithm the phase update of some flow is always based on all of its conflict flows. In Link Desynchronization, the update for the phase of f on some link l is only based on all flows traversing this exact link. As these flows traverse the same link as f, they also belong to its set of conflict flows. However, we see that this may only be a subset of all conflict flows of f, as f might conflict with further flows on a different link. Consequently, when updating the phase $\phi_{f,l}$ during desynchronization of link l, there may be less phases present on the desynchronization ring as compared to Relative Desynchronization. Having less different phases on our ring for desynchronization should consequently lead to possibly larger gaps between neighbouring phases on the desynchronization ring.

The Link Desynchronization algorithm is shown in Algorithm 5.4. As in the previously introduced approaches we first define a time interval in which we can start transmitting a flow so that it will always finish within the schedules cycle. Calculation of the latest starting time L works analogous to Algorithm 5.1. The time interval from 0 to the L is again mapped to the phase interval $[0, 1)$. Initially, we again randomly assign a starting phase ϕ_f to each flow f. For each flow f we then calculate $\phi_{f,l}$ for every link l on f's flow path. For each network link l we have a set of flows $\mathcal{F}_l$ that traverse this link. Only these flows are considered when desynchronizing link l.

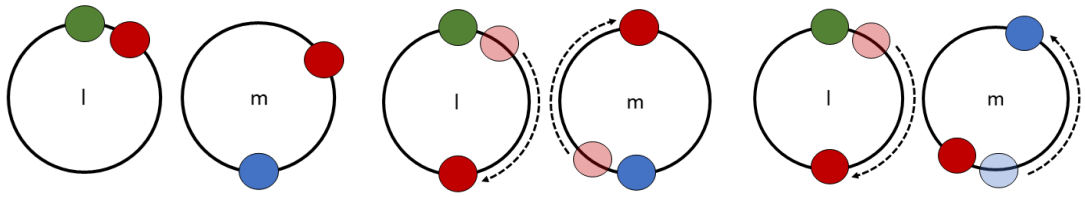
Instead of a global conflict graph, we only require for each network link the set of flows traversing this link (cf. Algorithm 5.4, line 10). We then iterate over all network links. For each link l, we update the phase $\phi_{f,l}$ once for every flow f from the set $\mathcal{F}_l$ of flows traversing l to resolve all local transmission conflicts. When updating $\phi_{f,l}$, we again elect the predecessor p and successor s of f from all flows in $\mathcal{F}_l$. In Link Desynchronization, predecessor and successor of a flow f on a link l are defined as the two flows in $\mathcal{F}_l$ with minimum $\Delta(\phi_{p,l}, \phi_{f,l})$ and $\Delta(\phi_{f,l}, \phi_{s,l})$, respectively. Consequently, we shift our phase $\phi_{f,l}$ to the mid-point of $\phi_{p,l}$ and $\phi_{s,l}$. We then calculate the phase difference between the old phase of flow f on link l and its updated value. Afterwards, we apply the same phase shift to the phases of f on all other links it traverses to keep the gap of size ϕ_{phd} between consecutive hops.

Algorithm 5.4 Link Desynchronization

```
1:  function LINK DESYNCHRONIZATION(max_iterations, threshold)
2:      F ← getAllFlows()
3:      L ← getAllLinks()
4:      maxShift ← 1
5:      initializeRandomPhases(F)
6:      n_iteration ← 1
7:      while maxShift > threshold and n_iteration < max_iterations do
8:          maxShift ← 1
9:          for all l ∈ randomize(L) do
10:             F_l ← getFlowsOnLink(l)
11:             for all f ∈ randomize(F_l) do
12:                 C ← F_l \ f
13:                 phaseShift ← updatePhase(f, C, l)
14:                 if phaseShift < maxShift then
15:                     maxShift ← phaseShift
16:
17: function UPDATEPHASE(f, C, l)
18:     s ← determineSuccessorOnLink(l, f, C)
19:     p ← determinePredecessorOnLink(l, f, C)
20:     if φ_{s,l} > φ_{p,l} then
21:         φ_new ← (φ_{s,l} + φ_{p,l})/2
22:     else
23:         φ_new ← (φ_{s,l} + φ_{p,l} + 1)/2
24:         if φ_new ≥ 1 then
25:             φ_new ← φ_new − 1
26:     φ_old ← φ_{f,l}
27:     φ_shift ← Δ(φ_old, φ_new)
28:     L_f ← getLinksOnFlowpath(f)
29:     for all m ∈ L_f do
30:         φ_{f,m} ← φ_{f,m} + φ_shift   mod 1
31:     absoluteShift ← |Δ(φ_old, φ_new)|
32:     return absoluteShift
```

After having updated all phases of a link l once, we move to the next link. Once we have updated all phases of all links once, we repeat the described procedure. We do this, until in one iteration no phase has been shifted by more than a predefined margin or if a predefined upper limit of iterations has been reached.

While after termination every flow f still has multiple phases assigned to it, we can obtain a concrete sending time by mapping the phase $\phi_f = \phi_{f,l}$ of the first link l on f's flow path to the $[0, L)$ interval. We do not need to map any other phases back to a concrete sending time, as the schedule is well defined by only transmission times of the source hosts under the assumption of *zero queueing*.



(a) Initial state **(b)** No Randomization **(c)** Randomization

Figure 5.7: Benefits of randomization in Link Desynchronization

Furthermore, in each iteration we process the links of the network in a randomized order (cf. Algorithm 5.4, line 9). The same holds for the phase updates of the flows traversing some link, where the order in which we update their phases will also be randomized (cf. Algorithm 5.4, line 11). Randomization will avoid Link Desynchronization to enter a state from which on no progress towards improvement of the desynchronization is made anymore. We will emphasize the benefits of randomization by giving a simple example. Assume we are given two network links l and m. Say link l is traversed by a flow f and a flow g. Link m is traversed by flow f and a flow h. Flow f further traverses link l before it traverses link m. The initial phases $\phi_{f,l}$, $\phi_{g,l}$, $\phi_{f,m}$ and $\phi_{h,m}$ for this example are illustrated in Figure 5.7a. The phases of flow f are represented by the red circles, the phases of flow g by the green circles, and the phases of flow h by the blue circles. Further assume that flow f is the only conflict flow of both flow g and flow h in the entire network. This implies that on any other link that flow g or h traverse, no further flow shares this link and thus no other phase is present on the desynchronization ring. Consequently, link l is the only link where we may update the phases of flow g, while link m is the only link where we may update the phases of flow h. Note that $\phi_{f,l} \neq \phi_{f,m}$ due to the per hop delay of the network.

First assume that we process link l before link m in every iteration, and on both links we update the phase of flow f first (cf. Figure 5.7b). When desynchronizing link l, we move $\phi_{f,l}$ to be as far away from $\phi_{g,l}$ as possible. After updating $\phi_{f,l}$, the link is already perfectly desynchronized, therefore the later update of $\phi_{g,l}$ will introduce no further changes. We then move to link m, where we update $\phi_{f,m}$ to be as far away as possible from $\phi_{h,m}$. After this update, the link is again perfectly desynchronized, thus the update of $\phi_{h,l}$ will also not introduce any changes. There are two things we can observe:

- The update to the phase $\phi_{f,l}$ of flow f made when desynchronizing link l is almost completely nullified by the later update to the phase $\phi_{f,m}$ of flow f when desynchronizing link m.
- The phases $\phi_{h,l}$ and $\phi_{g,m}$ of flow h and flow g will never be updated at any point throughout the algorithm.

If we were to execute another iteration with the current state of the phases, we see how after this iteration the phases will end up in an identical state, hindering progress towards improvement of the desynchronization. More precisely, after each iteration we end up with $\phi_{f,m} = 0$. The previous update on $\phi_{f,l}$ thus has absolutely no influence on the desynchronization results. Still, this update to $\phi_{f,l}$ will be applied in every iteration and prevent convergence of the desynchronization.

If we now randomize the order in which we desynchronize links and update phases, and again consider the initial state of Figure 5.7a, we quickly see that if at any point flow f does not update its phase first, but either flow g or flow h adjust their phase based on the current position of f, we immediately receive a perfectly desynchronized state (cf. Figure 5.7c). However, by pointing out the possibility that the phases of some flows might get stuck and not progress with increasing number of iterations, we also see that it is rather unlikely for the algorithm to converge.

This brings out a clear disadvantage of Link Desynchronization. If we are desynchronizing some link l, the updates on the respective phases are only based on the current desynchronization state of this link l. Updating $\phi_{f,l}$ of some flow f traversing l may however have a negative impact on the desynchronization state of some other link. Ideally, this negative impact will be reduced with an increasing number of executed iterations, as the desynchronization states of all links slowly converge to a globally desynchronized state. However, as convergence of Link Desynchronization cannot be guaranteed, we may end up terminating at the iteration limit right after some update on a link l has negatively impacted the desynchronization state of some other link m. As the iteration limit has been reached, there will be no chance to compensate for this in later desynchronization steps of link m. The rate of convergence of this algorithm will be further examined empirically in the evaluation chapter of this thesis.

5.4 Extended Link Desynchronization

Besides the mentioned concerns on convergence, the Link Desynchronization algorithm comes with one logical drawback, where some phases do not accurately represent the actual transmission time of the flows as they wrap around the $[0, 1)$ interval. We will further describe the concrete problem in detail below. Addressing it is not trivial and only possible by making compromises elsewhere. The resulting approach will be referred to as Extended Link Desynchronization. The modifications to the update method of the original Link Desynchronization algorithm made in this section can be seen in Algorithm 5.5. To address the problem briefly discussed above we establish a mapping between the $[0, 1)$ interval for desynchronization and the $[0, C)$ interval, where C is the length of the schedule cycle. While doing so we also need to ensure that every flow is still received in the same cycle in which it has started transmission. This was previously achieved by mapping from the $[0, 1)$ interval to the $[0, L)$ interval, where L was the latest starting time for any flow.

Before presenting the solution in detail, we first want to clearly describe the mentioned drawback of the Link Desynchronization algorithm presented in Section 5.3. Here we defined that a flow has multiple phases assigned to it, one for each network link along its flow path. The phase difference between two phases of a flow that belong to consecutive network links is defined by the per hop delay of the network $\mathcal{PHD}$ and its phase representation ϕ_{phd}.

Assume now the starting phase of some flow f with $\phi_f = 0.95$, that is the phase where the source host starts transmitting f over the first link, and $\phi_{phd} = 0.1$. Let link m be the second link on the flow path of f. We see that $\phi_{f,m} = \phi_f + \phi_{phd} = 1.05$. As we are desynchronizing phases on a ring, we wrap around the $[0, 1)$ interval and the phase $\phi_{f,m} = 1.05$ thus becomes $\phi_{f,m} = 0.05$.

Now assume that our schedule cycle has a length of 120 ms and our latest starting time L is 100 ms. With the mapping of Link Desynchronization between $[0, 1)$ and $[0, \mathrm{L})$, $\phi_f = 0.95$ will consequently be mapped back to 95 ms. The transmission on the second link m of f's flow path

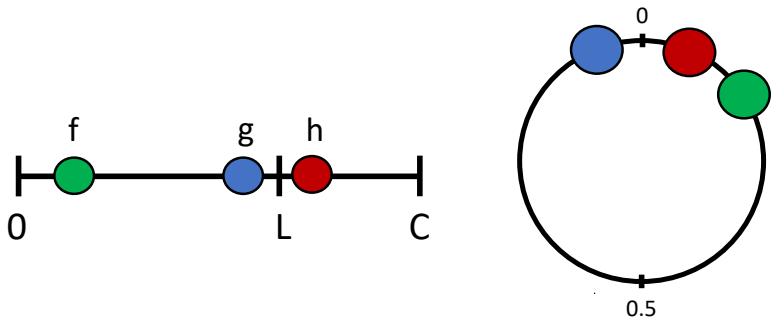


Figure 5.8: The actual sending time of a flow compared to its phase on the desynchronization ring

will begin at 105 ms. Obviously, the transmission time on the second link does not need to be smaller than L. This restriction only holds for the transmission over the first link of a flows flow path. The transmission time of 105 ms corresponds to the phase 1.05. This phase does not exist in the $[0, 1)$ interval, thus, as mentioned before, it instead is converted to 0.05. With the given mapping however, $\phi_{f,m} = 0.05$ would correspond to a sending time at 5 ms, but here no actual flow is being transmitted over link m. This introduces many phases at the start of the $[0, 1)$ interval, for which no actual transmission will take place.

We would like to emphasise that this behaviour does by no means break the desynchronization process of Link Desynchronization. All transmissions that take place between L and the end of the cycle will still be desynchronized, as all affected phases will still be close to each other in the $[0, 1)$ interval. From a logical standpoint, they are simply located at the wrong spot, as their actual transmission time is greater than L and thus cannot be accurately represented in the $[0, 1)$ interval. However, it still somewhat negatively impacts the Link Desynchronization algorithm, as these phases are still considered during the desynchronization process and may push away phases of other flows, even though their respective transmission times may not be close at all.

To illustrate this, we expand the example from before. Assume we have three flows f, g, and h traversing a link l. Let l be the first link on the flow path of flow f and g, but the second link on the flow path of flow h. Consequently, it holds that $\phi_f = \phi_{f,l}$ and $\phi_g = \phi_{g,l}$. In contrast, for flow h it holds that $\phi_{h,l} = \phi_h + \phi_{phd} \bmod 1$. Let ϕ_{phd} again be 0.1, and further let $\phi_{f,l} = 0.95$, $\phi_{g,l} = 0.1$ and $\phi_{h,l} = 0.05$ (cf. Figure 5.8 (right)). Figure 5.8 further visualizes the actual transmission times of the three flows on link l.

Here we observe that even though $\phi_{g,l}$ and $\phi_{h,l}$ are close to each other on the ring, their actual transmission times are very far apart from each other. While $\phi_{h,l} = 0.05$ would suggest that the transmission of flow h over link l takes place at the beginning of the cycle, this does not hold true because $\phi_h = 0.95 \neq \phi_{h,l} = 0.05$. As $\phi_h = 0.95$, the transmission is at the end of its cycle instead. Consequently, we have no actual transmission corresponding to $\phi_{h,l} = 0.05$. Still, the phase $\phi_{h,l}$ will push away nearby phases such as $\phi_{g,l}$ and $\phi_{f,l}$ during desynchronization and move them closer to other phases. Once the desynchronization terminates, the gaps between actual transmissions will thus be smaller than technically possible.

We see that the conversion between the $[0, 1)$ interval and the $[0, \mathrm{L})$ interval is the root of the problem. Instead, we can use the full cycle of the schedule and map the $[0, 1)$ interval to the $[0, \mathrm{C})$ interval, where C is the cycle length. Still, we need to guarantee that the actual starting phases ϕ_f

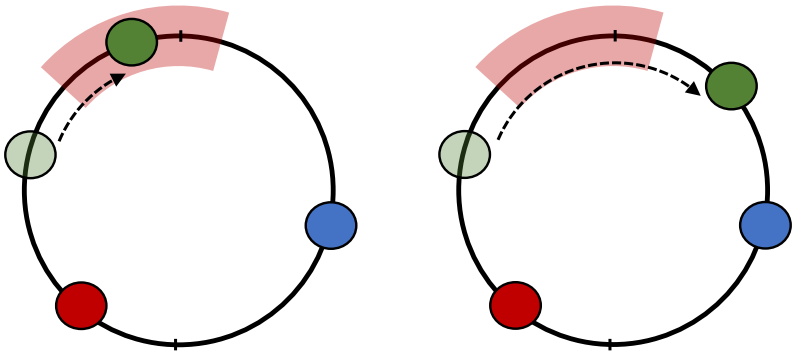


Figure 5.9: Violation interval of a flow on some link and the corresponding phase update

on the first link of a flow f's flow path are small enough that the flow can be fully received by its destination host before the current cycle has finished. Thus, we want to guarantee that for a flow f traversing a set of links $\mathcal{L}_f = \{l_1, l_2, \ldots, l_n\}$, where l_1 is the first link on f's flow path and l_n the last link on f's flow path, it holds true that

$$\phi_{f,l_i} < \phi_{f,l_{i+1}} \qquad \forall i < n$$

We still compute the latest starting time L as we did in previous sections, but this time convert it to phase representation with $\phi_{\mathrm{L}} = \mathrm{L}/C$. We have further seen in the section on Link Desynchronization that

$$\phi_{f,l_{i+1}} = \phi_{f,l_i} + \phi_{phd} \mod 1$$

If the starting phase ϕ_f of some flow f is smaller than ϕ_{L}, we can guarantee that the flow will be received by the destination host before the end of the current cycle. Consequently, we must avoid that $\phi_{\mathrm{L}} < \phi_f < 1$. This is equivalent to avoiding both $(\phi_{\mathrm{L}} + \phi_{phd}) < \phi_{f,l_2}$ and $\phi_{f,l_2} < \phi_{phd}$, as the difference between phases of a flow for two consecutive links is exactly ϕ_{phd}. If ϕ_{f,l_2} is larger than $\phi_{\mathrm{L}} + \phi_{phd}$, then it is guaranteed that ϕ_f is larger than ϕ_{L}. Analogously, if ϕ_{f,l_2} is smaller than ϕ_{phd}, this also implies that $\phi_{\mathrm{L}} < \phi_f < 1$.

From this information we can deduce that for every phase $\phi_{f,l}$ of some flow f, we can compute an interval in which we cannot allow $\phi_{f,l}$ to reside in. For a phase $\phi_{f,l}$ we will refer to this interval as the *violation interval* $\mathcal{V}_{f,l}$ of flow f on link l. For the starting phase ϕ_f this violation interval will always be $[\phi_{\mathrm{L}}, 1)$, or $[\phi_{\mathrm{L}}, 0)$ as 0 and 1 are identical phases on the ring. For every consecutive hop both sides of the violation interval are increased by ϕ_{phd}. More formally, the violation interval $\mathcal{V}_{f,l}$ for a phase $\phi_{f,l}$ is defined as follows:

Definition 5.4.1 (Violation interval)**.** $\mathcal{V}_{f,l} = [\ (\phi_{\mathrm{L}} + \eta_{f,l} * \phi_{phd})\ ,\ (\eta_{f,l} * \phi_{phd})\ )$

Here, $\eta_{f,l}$ is defined as in previous sections. Intuitively, once some phase $\phi_{f,l}$ of some flow f on a link l is updated and falls in its violation interval, we can simply shift it to the end of the violation interval instead. By definition of the violation interval, we are guaranteed that all other phases of flow f on other links also are at the end of their respective violation interval, if they are shifted by the same distance. However, when examining results using this approach, compared to the

unmodified Link Desynchronization algorithm we noticed that this heavily impacts the distribution of all phases in our network and a majority of them will reside in the lower regions of our [0,1) interval. This behaviour is unwanted for desynchronization.

Instead, we let the total phase shift of the update that originally placed the phase in the violation interval influence the new phase. Say $\phi_{f,l} = 0.8$ and its violation interval is $[0.85, 0.1)$. Assume now an update to $\phi_{f,l}$ results in $\phi_{f,l} = 0.95$ (cf. Figure 5.9 (left)). This would shift the phase by 0.15 in clockwise direction on the ring and place it within its violation interval. Instead of simply moving it to 0.1, the end of its violation interval, we move it by the determined 0.15 in clockwise direction, however this time we apply this shift on the interval $[0.1, 0.85)$, which is the complement of the violation interval. By doing this, we simply increase the shift of the initial update by the size of the violation interval. On the defined interval, shifting $\phi_{f,l} = 0.8$ by 0.15 in clockwise direction results in the new phase $\phi_{f,l} = 0.2$ (cf. Figure 5.9 (right)). If the phase shift of an update is larger than 0.5, we see that this actually corresponds to a backwards movement on the ring, as the new phase can be reached quicker in counter-clockwise direction. Say now a phase $\phi_{f,l} = 0.2$ with violation interval $[0.85, 0.1)$ is updated to $\phi_{f,l} = 0.95$ and thus falls in its violation interval. The absolute phase shift is 0.25 in counter-clockwise direction. We apply the same concept as for shifts in clockwise direction and set the new phase as $\phi_{f,l} = 0.7$. For phase shifts smaller than 0.5, and thus a shift in clockwise direction, the new phase $\phi_{f,l}$ is given by

$$\phi_{f,l} \leftarrow \mathcal{V}_{f,l}(u) + (\phi_{shift} - (\mathcal{V}_{f,l}(l) - \phi_{old}))$$

where $\mathcal{V}_{f,l}(l)$ is the lower end of flow f's violation interval on link l, $\mathcal{V}_{f,l}(u)$ the upper end of this violation interval, ϕ_{old} the old phase of flow f on link l before the update and ϕ_{shift} the phase shift of the initial update placing $\phi_{f,l}$ within its violation interval (cf. Algorithm 5.5, line 16). Analogously, for phase shifts larger than 0.5, and thus a shift in counter-clockwise direction, the new phase $\phi_{f,l}$ is given by

$$\phi_{f,l} \leftarrow \mathcal{V}_{f,l}(l) - (\phi_{shift} - (\phi_{old} - \mathcal{V}_{f,l}(u)))$$

This modification yields a phase distribution similar to the distribution of Link Desynchronization, as it does not blindly move all violating phases to the immediate start of the ring. While solving the logical flaw of Link Desynchronization, we observe that when moving a violating phase to some new point outside of its violation interval, this update does not take the neighbourhood around the new phase into account. Immediately after the update, we thus might impair our desynchronization of the current link for a short time. This will eventually balance out, as other phases adapt to the update on the violating phase. However, as we previously outlined the unlikelihood of convergence for our Link Desynchronization approach, this might still lead to termination in a suboptimal state if an update on a violating phase is applied just before the iteration limit is reached.

5.5 Ordered Desynchronization

The fifth and last approach presented in this section moves away from the classic desynchronization idea based on [DRN07] and instead draws inspiration from the general idea of slotting used by M-DESYNC [KW09]. Here, we determine a set of slots among which the network flows will be distributed in a way that as many collisions as possible are avoided.

Algorithm 5.5 Adaptation for Extended Link Desynchronization

```
1:  function UpdatePhase(f, C, l)
2:      s ← determineSuccessorOnLink(l, f, C)
3:      p ← determinePredecessorOnLink(l, f, C)
4:      if φ_{s,l} > φ_{p,l} then
5:          φ_new ← (φ_{s,l} + φ_{p,l})/2
6:      else
7:          φ_new ← (φ_{s,l} + φ_{p,l} + 1)/2
8:          if φ_new ≥ 1 then
9:              φ_new ← φ_new − 1
10:     (V_{f,l}(l), V_{f,l}(u)) ← violationInterval(φ_{f,l})
11:     φ_old ← φ_{f,l}
12:     φ_shift ← Δ(φ_old, φ_new)
13:
14:     if (φ_new > V_{f,l}(u)) or (φ_new < V_{f,l}(l)) then
15:         if φ_shift < 0.5 then                          // Clockwise
16:             φ_new ← V_{f,l}(u) + (φ_shift − (V_{f,l}(l) − φ_old))
17:         else                                           // Counter-clockwise
18:             φ_new ← V_{f,l}(l) − (φ_shift − (φ_old − V_{f,l}(u)))
19:     φ_shift ← Δ(φ_old, φ_new)
20:     L_f ← getLinksOnFlowpath(f)
21:     for all m ∈ L_f do
22:         φ_{f,m} ← φ_{f,m} + φ_shift mod 1
23:     absoluteShift ← |Δ(φ_old, φ_new)|
24:     return absoluteShift
```

When introducing Relative Desynchronization, we have seen that when desynchronizing Time-Sensitive Networks it is important to consider both the path length up to a conflict link and the actual starting time of the flow. In particular, we have seen that if we desynchronize the starting times of two flows f_1 and f_2 but send the flow with the longer path length up to the conflict link first, they might still arrive simultaneously at the conflict link. However, if we send the flow with the smaller path length up the first conflict link first, we can observe that the two flows f_1 and f_2 cannot collide at any link in the topology anymore, as the later flow will not be able to catch up. This of course is based on our assumption of uniform network delays and flow sizes.

For improved readability we first define notations for these pairwise relations between two flows. We write $f_1 \succ f_2$ (f_1 *succeeds* f_2) if the two flows conflict on at least one link l and the number of hops of f_1 up to this link l is greater than the number of hops of f_2 up to this link l. Analogously, we write $f_1 \prec f_2$ (f_1 *precedes* f_2) if the two flows conflict on at least one link l and the number of hops of f_1 up to this link l is smaller than the number of hops of f_2 up to this link l. It is important to remember that if two flows f_1 and f_2 conflict on more than just a single network link, we have shown that the difference in path length up to any of the conflict links between the two flows always remains consistent due to our assumptions of shortest path routing. Furthermore, there is only a single path connecting any two nodes in a tree topology. Thus, if two flows conflict with each other and their paths diverge after some conflict link, their remaining paths up to the respective destination node can never intersect again, as this would mean there are two different paths to this

new conflict link. From this we follow that if two flows have a conflict at some network link, all other conflict links of these two flows must be consecutive network links of their flow paths. As a result, if $f_1 \prec f_2$ holds for some conflict link of f_1 and f_2, it will also hold for every other conflict link. Therefore, we can guarantee that f_1 and f_2 can never collide if f_1's starting time is smaller than f_2's starting time.

We further write $f_1 \equiv f_2$ (f_1 *is equivalent to* f_2) if they conflict on at least one link l and their path length up to this link l is equal. We can see that this is in fact the most critical conflict between two flows. Obviously, sending both flows simultaneously will inevitably lead to a collision between f_1 and f_2. However, just sending one flow before the other does not guarantee a collision-free transmission, as the interval between both starting times plays a major role. We can only avoid a collision, if the difference in starting times between both flows is larger than the transmission delay plus the inter frame gap. The actual order in which the two flows are eventually sent is not relevant if the previously mentioned requirements are met.

Lastly, we write $f_1 \parallel f_2$ (f_1 *is independent of* f_2) if the two flows f_1 and f_2 do not conflict with each other at all.

The algorithm presented in the following can be divided into two steps. In the first step we want to order all flows $\mathcal{F}$ of the network so that for any pair of flows f_1 and f_2 the flow f_1 is ordered before f_2, if it holds true that $f_1 \prec f_2$. Thus, we will further refer to the presented algorithm as Ordered Desynchronization. If we later send the flows in the previously determined order, we are guaranteed that all pairs of flows f_1 and f_2, for which either $f_1 \prec f_2$ or $f_2 \prec f_1$ holds true, can never collide at any link of the network for the reasons described above.

In the second step of the algorithm, we then desynchronize the flows by determining suitable sending time intervals between consecutive flows in the derived order. This is especially catered towards pairs of two flows f_1 and f_2 with the relation $f_1 \equiv f_2$, for which the avoidance of collisions could not be guaranteed in the previous step. As we have previously outlined that this is the most critical relationship between two flows, we need to make sure that the sending time interval between any two flows with an *equivalent*-relation is large enough to avoid a collision. For some topology types, as further discussed in the remainder of this section, this process can even enable the calculation of a collision-free schedule.

While we are using the terms *order* and *ordering*, we want to explicitly make a clear distinction to the mathematical concept of an order, as the latter requires transitivity to hold. However, for the relations introduced above, transitivity cannot be guaranteed. Assume three flows f_1, f_2, and f_3. If $f_1 \prec f_2$ and $f_2 \prec f_3$, this does not imply that $f_1 \prec f_3$. In fact, f_1 and f_3 can also be completely independent ($f_1 \parallel f_3$), as is demonstrated in Figure 5.10. In the following section we will often simplify multiple relations such as the ones listed above to a single term. For this example, we end up with the term $f_1 \prec f_2 \prec f_3$, so remember that this notation does not imply any relation between f_1 and f_3.

Having described the basic structure of the presented algorithm, we now want to formally define the criteria that the order we want to obtain has to fulfil. Here, we want to obtain an ordered set $O = \{f_1, f_2, \ldots, f_n\}$ containing all flows $\mathcal{F}$ of the network, for which it holds that

$$\neg(f_i \succ f_j) \qquad \forall f_i, f_j \in O \; with \; i < j$$

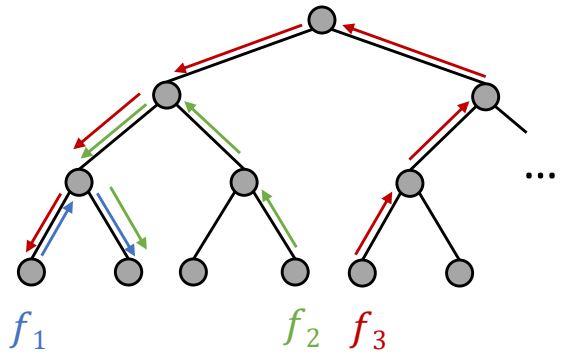


Figure 5.10: Examining transitivity of the *succeeds*- and *precedes*-relations

Especially the order of two flows f_i and f_j with $f_i \equiv f_j$ is arbitrary according to this definition. More so, they are not even required to be ordered *back-to-back*. Assume the four flows f_1, f_2, f_3, and f_4 with $f_1 \equiv f_2$, $f_2 \prec f_3$, and $f_3 \equiv f_4$. We see that the ordered set $O = \{f_4, f_2, f_1, f_3\}$ does not violate any of the criteria for the desired order defined above, as the only requirement is to order f_2 before f_3. Obviously, by the nature of the *equivalent*-relation these flows are more likely to collide if their sending times are very close to each other. However, even if they are ordered directly back-to-back, we can avoid a collision between these flows in the second step of the algorithm, where the sending time interval between consecutive flows in the obtained ordered set is determined.

The question is now if an order fulfilling the listed criteria can always be calculated. Here, we will especially examine the balanced and imbalanced tree topologies introduced in Chapter 4. Assume three flows f_1, f_2, and f_3 with $f_1 \prec f_2$, $f_2 \prec f_3$, and $f_3 \prec f_1$ ($f_1 \prec f_2 \prec f_3$). It is clear to see, that given these relations we cannot obtain an order fulfilling the listed criteria. If we order f_1 before f_3, this violates the criteria because $f_3 \prec f_1$. Vice versa, if we order f_3 before f_1, then we can never fulfil both $f_1 \prec f_2$ and $f_2 \prec f_3$. We call such a constellation of flows a *cyclic relationship*.

Definition 5.5.1 (Cyclic relationship)**.** A cyclic relationship is a subset $\mathcal{F}' = \{f_1, f_2, \ldots, f_k\}$ of all network flows $\mathcal{F}$ for which it holds that $f_1 \prec f_2 \prec \cdots \prec f_k \prec f_1$

In the context of the Ordered Desynchronization algorithm, the presence of cyclic relationships prevents the calculation of the desired order. In these cases, the algorithm is only able to calculate a best-effort approximation of the desired order, which violates some of the listed criteria. Therefore, sending the network flows in the determined order will thus avoid less collisions.

Theorem 1. *In a balanced tree topology, a cyclic relationship $\mathcal{F}'$ cannot exist*

Proof. To proof the claim of Theorem 1, we first examine how and when a relation of the form $f_i \prec f_j$ occurs in a balanced tree topology.

In our system model the host nodes of tree topologies are exclusively leaf nodes. Consequently, each flow will have a sub-path traversing the tree upwards in the direction of the tree root, and a sub-path traversing the tree downwards in the direction of the destination host. We refer to the

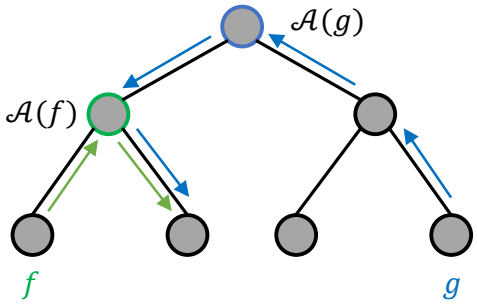


Figure 5.11: Apex nodes $\mathcal{A}(f)$ and $\mathcal{A}(g)$ of two flows f (green) and g (blue) in a tree topology

lowest-level node in the tree that a flow f traverses as its apex node $\mathcal{A}(f)$. Here we consider the tree root to be on the lowest level of a tree topology. Figure 5.11 visualizes the apex node of two flows f and g in a balanced tree topology.

It is obvious that two flows can only conflict if at some shared full duplex link of the balanced tree topology they both are either on their upwards sub-path or on their downwards sub-path. If this is not the case, both flows traverse the link in different directions and there can be no transmission conflict on full duplex network links. First assume that the conflict link of two flows f_1 and f_2 lies on both their upwards sub-paths. We observe that in a tree topology a flow will only traverse exactly one link per tree level on its upwards sub-path. The distance from a leaf node to some tree level is equal for all leaf nodes in a balanced tree, as all leaf nodes reside on the same level. Thus, if two flows conflict on a link while still traversing the tree upwards, their length up to the conflict link will be equal and it holds that $f_1 \equiv f_2$.

Now assume that the conflict link of f_1 and f_2 lies on the downwards sub-path of both f_1 and f_2. This means that they already traversed their apex nodes $\mathcal{A}(f_1)$ and $\mathcal{A}(f_2)$ and are now moving downwards in direction of their respective destination host.

Say $\mathcal{A}(f_1)$ and $\mathcal{A}(f_2)$ reside on the same level of the tree. Under the assumption that f_1 and f_2 conflict on a link on their downwards sub-paths, this immediately implies that $\mathcal{A}(f_1) = \mathcal{A}(f_2)$. This is because in tree topologies we have a fan-out from the root node towards the leaf nodes. Thus, two distinct paths diverging at some node of the tree and pointing towards the leaf nodes can never merge again. Consequently, if $\mathcal{A}(f_1) = \mathcal{A}(f_2)$, then there can be no two distinct paths from $\mathcal{A}(f_1)$ to the conflict link and from $\mathcal{A}(f_2)$ to the conflict link if we only consider links that point towards the leaf nodes of the tree. If $\mathcal{A}(f_1) = \mathcal{A}(f_2)$, then it must also hold that $f_1 \equiv f_2$, as we can apply the same reasoning as we did for conflicts on the upwards sub-paths. If $\mathcal{A}(f_2)$ resides on a higher level than $\mathcal{A}(f_1)$ and is thus closer to the leaf nodes, then $f_1 \prec f_2$ still cannot hold. The path from f_2's source host to $\mathcal{A}(f_2)$ is shorter than the path from f_1's source host to $\mathcal{A}(f_1)$. Analogously, the path from $\mathcal{A}(f_2)$ to the conflict link of f_1 and f_2 is shorter than the path from $\mathcal{A}(f_1)$ to the conflict link. As a result, it holds that $f_1 \succ f_2$. Consequently, $f_1 \prec f_2$ holds if and only if $\mathcal{A}(f_1)$ resides on a higher level than $\mathcal{A}(f_2)$.

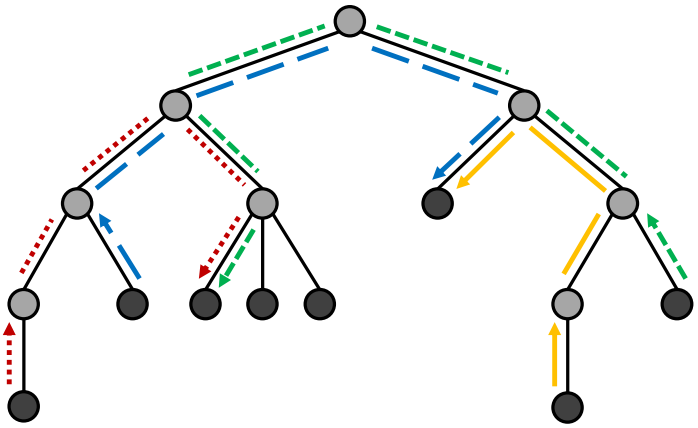


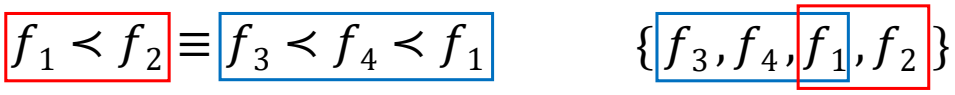
Figure 5.12: Exemplary cyclic relationship of four flows f_1 (blue), f_2 (red), f_3 (green) and f_4 (yellow) in an imbalanced tree topology. Host nodes of the tree are coloured dark.

Now consider the cyclic relationship $\mathcal{F}'$ with $f_1 \prec f_2 \prec f_3 \prec f_4 \prec f_1$. Here, $\mathcal{A}(f_1)$ must reside on a higher level than $\mathcal{A}(f_2)$. $\mathcal{A}(f_2)$ in turn must reside on a higher level than $\mathcal{A}(f_3)$, and so on. This implies that $\mathcal{A}(f_1)$ must reside on a higher level than $\mathcal{A}(f_4)$. This is contradictory to the relation $f_4 \prec f_1$, which requires $\mathcal{A}(f_4)$ to reside on a higher level than $\mathcal{A}(f_1)$. Thus, a cyclic relationship cannot exist in a balanced tree topology. □

In an imbalanced tree however, we cannot guarantee the absence of *cyclic relationships*, as the leaf nodes do not necessarily reside on the same level of the tree. This implies that some flows can reach nodes at some level of the tree in fewer hops than other flows, depending on the tree level their source node is located on. This invalidates the proof described above. We can demonstrate the possibility to end up with cyclic relationships in an imbalanced tree by giving a simple example, which is also depicted in Figure 5.12. It shows an imbalanced tree with five tree levels. Four different flows f_1 (blue), f_2 (red), f_3 (green) and f_4 (yellow) are highlighted. We see that $f1 \prec f2$, $f2 \prec f3$, $f3 \prec f4$ and $f4 \prec f1$ resulting in the cyclic relationship $\mathcal{F}'$ with $f_1 \prec f_2 \prec f_3 \prec f_4 \prec f_1$. Consequently, for imbalanced trees we cannot guarantee that we can order all flows without violating some of the criteria. In this case, computation of a collision-free schedule is very unlikely.

To briefly summarize the recent paragraphs, we have shown that for a set of flows with a cyclic relationship no order fulfilling the criteria introduced at the beginning of this chapter can be obtained. We have further shown that for imbalanced tree topologies we cannot guarantee the absence of cyclic relationships. As a result, it is very unlikely that the Ordered Desynchronization algorithm, which will be presented in detail at a later point in this chapter, computes a collision-free schedule to any scheduling problem on imbalanced tree topologies. For balanced trees we have proven that cyclic relationships cannot exist. Yet this is no definite proof that on balanced tree topologies the desired order can always be obtained, as cyclic relationships might not be the only hindering factor.

Assume three flows f_1, f_2, and f_3 with $f_1 \prec f_2$, $f_2 \equiv f_3$, and $f_3 \prec f_1$. As before, we simplify the three relations to the term $f_1 \prec f_2 \equiv f_3 \prec f_1$. At first glance this example looks very similar to a cyclic relationship. However, consider the following ordered set $O = \{f_3, f_1, f_2\}$. We see how this



(a) Exemplary semi cyclic relationship **(b)** Resulting order after the split

Figure 5.13: Splitting a semi cyclic relationship to obtain an order

order does not violate any of the relations listed above. f_1 is ordered before f_2 as required, f_3 is ordered before f_1 as required, and the order of f_3 and f_2 is arbitrary because $f_2 \equiv f_3$. We call such a constellation of flows a *semi cyclic relationship*.

Definition 5.5.2 (Semi cyclic relationship)**.** A semi cyclic relationship is a subset $\mathcal{F}' = \{f_1, f_2, \ldots, f_k\}$ of all flows $\mathcal{F}$ for which it holds true that

$$\neg(f_j \succ f_i) \wedge \neg(f_j \parallel f_i) \qquad \forall i, j \; with \; j = i + 1 \leq k$$

This definition is very similar to a cyclic relationship (cf. Definition 5.5.1). The key difference is that it does not require $f_i \prec f_j$ to hold for two consecutive flows in the semi cyclic relationship, but instead also allows for $f_i \equiv f_j$.

A semi cyclic relationship can occur in any of the considered topologies, even in a balanced tree. Thus, it remains to show that the presence of such semi cyclic relationship will not prohibit the calculation of the desired order.

Assume we are given the semi cyclic relationship $\mathcal{F}'$ with $f_1 \prec f_2 \equiv f_3 \prec f_4 \prec f_1$. This semi cyclic relationship thus only requires the starting time of f_1 to be smaller than f_2's starting time, f_3's starting time to be smaller than f_4's starting time, and f_4's starting time to be smaller than f_1's starting time. As outlined, the relation $f_2 \equiv f_3$ allows us to order the two flows f_2 and f_3 arbitrarily. The order of all flows on the left side of the *equivalent*-relation is well defined by the relation $f_1 \prec f_2$. The same holds for all flows on the right side of the *equivalent*-relation. Thus, we can always deduce an order by *splitting* the semi cyclic relationship at the *equivalent*-relation (cf. Figure 5.13a). Here, we order all flows on the left side of the *equivalent*-sign in a semi cyclic relationship according to their relations (either $\succ$ or $\prec$). We do the same for all flows on the right side of the *equivalent*-sign in a semi cyclic relationship. A split on the given example gives us the two ordered sets $\{f_1, f_2\}$ and $\{f_3, f_4, f_1\}$. We then append the ordered set of all flows on the left of the *equivalent*-relation to the set of flows on the right side. This results in the set $\{f_3, f_4, f_1, f_2\}$ (cf. Figure 5.13b). We see that this set satisfies all criteria of the desired order.

If a semi cyclic relationship contains multiple relations of the form $f_i \equiv f_j$, we can apply the same approach at an arbitrary *equivalent*-sign in the semi cyclic relationship. For the semi cyclic relationship $\mathcal{F}'$ with $f_1 \prec f_2 \equiv f_3 \prec f_4 \equiv f_5 \prec f_1$, we obtain the ordered set $\{f_3, f_4, f_5, f_1, f_2\}$ if we split at the relation $f_2 \equiv f_3$. Likewise, we obtain the ordered set $\{f_5, f_1, f_2, f_3, f_4\}$ if we split at the relation $f_4 \equiv f_5$. In both cases, all criteria are satisfied.

Consequently, for a balanced tree topology we should be able to always compute an ordered set O that satisfies all listed criteria.

Based on this knowledge, we now discuss the Ordered Desynchronization algorithm (cf. Algorithm 5.6), whose broad structure has been outlined before, in detail. First, we introduce *buckets*. A *bucket* acts as a container and can store any number of flows. We refer to the flows stored in some bucket B as $\mathcal{F}(B)$. We also define a *bucket list* as an ordered list of buckets. The goal of the algorithm now is to distribute all flows among a set of buckets and order these buckets in a bucket list. The algorithm aims to fulfil the following properties for the calculated bucket list:

1. No bucket B contains two or more flows that conflict with each other. Thus, for no f_i and $f_j \in \mathcal{F}(B)$ it holds that $f_i \prec f_j$, $f_i \equiv f_j$ or $f_i \succ f_j$.
2. For no flow f_i in any bucket B there exists a bucket A that comes before B in the bucket list and contains a flow f_j, for which it holds that $f_i \prec \mathrm{f}_j$.
3. For no flow f_i in any bucket B there exists a bucket C that comes after B in the bucket list and contains a flow f_j, for which it holds that $f_i \succ f_j$.

These properties correspond to the criteria of the previously discussed, desired order. Given a bucket list fulfilling these three properties, all flows of the same bucket may be sent in an arbitrary order, as they are independent of another and can never collide. Moreover, all flows of the same bucket can even be sent simultaneously.

Initially, we create a new bucket list containing a single bucket B and add all flows $\mathcal{F}$ of the network to this bucket. We then iterate over all pairs (f_i, f_j) of flows in $\mathcal{F}(B)$ and examine their pairwise relations (cf. Algorithm 5.6, line 10). By doing so, we want to determine the set of flows that eventually need to be removed from $\mathcal{F}(B)$ and moved up into a new bucket. If for a pair (f_i, f_j) it holds that $f_i \prec f_j$, we mark f_j to be moved up to a new bucket. If it holds, that $f_i \succ f_j$, we mark f_i to be moved into a new bucket instead.

After iterating over all flow pairs, all flows succeeding at least one other flow in $\mathcal{F}(B)$ are now marked. We can now remove all marked flows from bucket B and thus also from $\mathcal{F}(B)$ (cf. Algorithm 5.6, line 17). The set of marked flows in this previous step is denoted by $\mathcal{M}$.

At this point, we have not yet considered the relations of the form $f_i \equiv f_j$. For this, we again iterate over all pairs (f_i, f_j) of flows remaining in $\mathcal{F}(B)$. Note that $\mathcal{F}(B)$ now only contains the flows that have not been marked already in the previous step. If it now holds that $f_i \equiv f_j$, we need to move up at least one of the flows f_i or f_j from bucket B to fulfil the first property of a bucket list introduced above. As the actual order between two equivalent flows is arbitrary, we introduce a tiebreaker rule to stay consistent over the whole ordering process. In this case, we mark the flow with the larger *flow id*.

After iterating over all pairs of flows remaining in $\mathcal{F}(B)$, we again remove all marked flows from $\mathcal{F}(B)$ (cf. Algorithm 5.6, line 25). The set of marked flows in this second step is now denoted by $\mathcal{M}'$. At this point, it is guaranteed that all flows that were not marked by now and thus remain in $\mathcal{F}(B)$ do not conflict with any other flow in bucket B. Furthermore, it is also guaranteed that for no two flows $f_i \in \mathcal{M} \cup \mathcal{M}'$ and $f_j \in \mathcal{F}(B)$ it holds that $f_i \prec f_j$. We thus create a new bucket C and append it to the bucket list. We then add all flows previously removed from $\mathcal{F}(B)$ to this new bucket by setting $\mathcal{F}(C) = \mathcal{M} \cup \mathcal{M}'$ (cf. Algorithm 5.6, line 28).

We then repeat the same procedure described above, only that we are now iterating over all pairs of flows in $\mathcal{F}(C)$, and we are marking them to later be moved to a bucket D. We do this, until in one step either no flow has been moved to the succeeding bucket (cf. Algorithm 5.6, line 30), or if the current bucket to iterate over initially only contains a single flow (cf. Algorithm 5.6, line 32).

In every step we mark and remove all flows that succeed at least one other flow first, and only afterwards mark and remove flows with *equivalent*-relations via tiebreaker rule, because this ensures that we receive the split discussed previously when introducing semi cyclic relationships. We will demonstrate this with a simple example. Given the semi cyclic relationship $\mathcal{F}'$ with $f_1 \prec f_2 \equiv f_3 \prec f_4 \prec f_1$. Assume f_3's *flow id* is larger than f_2's *flow id*. If we would process all relation types simultaneously, we might run into a problem when the flow pair (f_2, f_3) is the first flow pair processed. Here, f_3 would be marked to be moved up to the succeeding bucket via tiebreaker rule. However, as for all other flows f_j there exists at least one other flow f_i for which $f_i \prec f_j$ holds, we would end up marking all flows to be moved up and not making any progress with the ordering.

By first considering pairs with *precedes*- and *succeeds*-relations only, we avoid this problem completely. It is obvious, that any flow that succeeds at least one other flow must inevitably be moved up to the next bucket. In our example above we would thus move up f_1, f_2 and f_4, leaving only f_3 in the lower bucket. This will eventually yield the order f_3, f_4, f_1, f_2 and is thus splitting the *semi cyclic relationship* exactly at the *equivalent*-relation $f_2 \equiv f_3$.

We further observe that the ordering process eventually terminates, if in every step at least one flow remains in the lower bucket. If in one step all flows are moved to a next bucket, the set of flows to be ordered does not change for the following step and we will end up in an endless loop. For a balanced tree, we can guarantee that the latter cannot occur, as we have shown that no *cyclic relationships* exist in this topology. For other topology types where this prove does not apply, we thus need to explicitly ensure termination.

For this, we modify the procedure described above by simply stopping to mark flows to be moved to the succeeding bucket, if all but a single flow have already been marked (cf. Algorithm 5.6, line 15). Thus, in each step, at least one flow remains in the lower bucket and the algorithm will eventually terminate, as the number of flows in the current bucket being processed decreases with each step. We further argue, that as the flow remaining in the lower bucket was the last flow that has not been marked, the result will most likely violate comparably few properties of a bucket list.

After obtaining an ordered bucket list through the described ordering process, we now need to assign concrete starting times to all flows. As outlined above, flows of the same bucket are independent of another and we thus assign all flows of the same bucket the same starting time. We also discussed how two conflicting flows cannot collide if the flow with the shorter path length up to the conflict link is sent first. If we calculated an ordered bucket list, it is guaranteed that for some flow f in a bucket B there is no conflicting flow g in any bucket coming after B in the bucket list with a shorter path up to the conflict link. Thus, if we send all flows of some bucket before sending any flow of a succeeding bucket, we are also guaranteed to avoid collisions between these flows.

Consequently, a very easy approach to schedule the flows is to uniformly distribute the buckets over the cycle of the schedule. Obviously, we still must ensure that all flows are received within the same cycle they were sent in. The calculation of the latest starting time L analogous to Algorithm 5.1 thus remains essential. Given a latest starting time L of 10 ms and a bucket list consisting of six buckets this would result in a 2 ms interval between two consecutive buckets.

Algorithm 5.6 Ordered Desynchronization

```
1: function ORDERED DESYNCHRONIZATION(cycle)
2:     F ← getAllFlows()
3:     BL ← createEmptyBucketList()
4:     B ← newBucket()
5:     F(B) ← F
6:     BL.append(B)
7:     while true do
8:         M ← {}                                  // First set of marked flows (≻ and ≺)
9:         B ← BL.getLastBucket()
10:        for all pairs (f_i, f_j) ∈ F(B) do
11:            if f_i ≺ f_j then
12:                M ← M ∪ f_j
13:            else if f_j ≺ f_i then
14:                M ← M ∪ f_i
15:            if |M| = |F(B)| − 1 then
16:                break
17:        F(B) ← F(B) \ M
18:        M' ← {}                                 // Second set of marked flows (≡)
19:        for all pairs (f_i, f_j) ∈ F(B) do
20:            if f_i ≡ f_j then
21:                if f_i.id ≺ f_j.id then
22:                    M' ← M' ∪ f_j
23:                else
24:                    M' ← M' ∪ f_i
25:        F(B) ← F(B) \ M'
26:        C ← newBucket()
27:        if |M ∪ M'| > 0 then
28:            F(C) ← M ∪ M'
29:            BL.append(C)
30:        else
31:            break
32:        if |F(C)| = 1 then
33:            break
34:
35:    D ← computeDistances(BL)
36:    D ← invertDistances(D)
37:    TD ← computeTotalDist(D)
38:    CD ← cumulativeDistances(D)
39:    L ← latestStart(cycle)
40:    for all buckets B ∈ BL do
41:        S(B) = (CD(B) ÷ TD) ∗ L
```

However, this very simple approach does not always yield optimal results. To show this, we examine two consecutive buckets B and C and their respective sets of flows $\mathcal{F}(B)$ and $\mathcal{F}(C)$. Let $\mathcal{F}(B) \times \mathcal{F}(C)$ be the cross-product of $\mathcal{F}(B)$ and $\mathcal{F}(C)$. $\mathcal{F}(B) \times \mathcal{F}(C)$ thus contains the set of unique pairings (f_i, f_j) of flows f_i from $\mathcal{F}(B)$ and flows f_j from $\mathcal{F}(C)$. If for all pairs (f_i, f_j) it holds that $f_i \parallel f_j$, we observe that we could theoretically send all flows of both buckets B and C simultaneously without getting any collisions. Further, the same holds even if for all pairs (f_i, f_j) it holds true that $f_i \prec f_j$, as we assume uniform transmission delays and flow sizes. However, while avoiding collisions, the actual difference in transmission times at the conflict link may be very small in this case. If for at least one pair it holds that $f_i \equiv f_j$, we immediately see that sending the flows of bucket B and C simultaneously would result in f_i and f_j colliding, as their path length up to the conflict link is equal. In fact, f_i and f_j will always collide, if the interval between the starting times of bucket B's flows and Bucket C's flows is smaller than transmission delay plus the inter frame gap. As a result, it seems logical to have a larger difference in starting times between flows f_i and f_j with $f_i \equiv f_j$ than for flows f_k and f_l with $f_k \prec f_l$.

Consequently, instead of distributing the buckets uniformly over the cycle, it is better to adjust the interval between two buckets based on the relations between their respective flows. For two conflicting flows f_i and f_j, we define the distance from flow f_i to flow f_j as the difference in path length up to their conflict link. This distance is considered positive if $f_i \prec f_j$. However, if $f_j \prec f_i$, then we consider the distance from f_i to f_j to be negative. In case of $f_i \equiv f_j$, the distance will be zero.

Further, we define the distance between a bucket B and its succeeding bucket C as the minimum distance from a flow f_i to a flow f_j, where $f_i \in \mathcal{F}(B)$ and $f_j \in \mathcal{F}(C)$. Thus, if for at least one pair (f_i, f_j) it holds that $f_i \equiv f_j$, the distance between B and C is zero. If a bucket list fulfils all listed properties, we can further guarantee that the distance from any bucket B to its consecutive bucket in the bucket list will never be negative. Intuitively, the larger the distance between two buckets, the less we need to space out the starting times of their flows. Likewise, if the distance between two buckets B and C is zero, we want to ensure that the difference between the sending times of all flows from $\mathcal{F}(B)$ and all flows from $\mathcal{F}(C)$ is as large as possible.

To compute starting times for our buckets we therefore start by calculating the distances between each bucket and its directly succeeding bucket in the bucket list (cf. Algorithm 5.6, line 35). If we are not desynchronizing a balanced tree, we have shown that an order cannot always be obtained, and we only try to derive the best possible approximation of it. However, as the derived order may violate some of the desired properties, the distance between two consecutive buckets can also be negative in this case, as for a flow f_1 in the lower bucket and a flow f_2 in the upper bucket it may hold that $f_1 \succ f_2$. If the distance between two buckets is negative, we simply assume the distance to be zero instead.

To make the process of calculating the intervals between the buckets more intuitive, we take the minimum distance between any two consecutive buckets and the maximum distance between any two consecutive buckets and then invert the distances between all buckets accordingly. If we have a minimum distance of zero and a maximum distance of five, all buckets with distance five to its succeeding bucket will now be considered to have a distance of zero and all buckets which previously had a distance of four to its succeeding bucket will now be considered to have a distance of one, and so on. This is a more intuitive representation, as it now reflects that two consecutive

buckets with a large distance need to be spaced out farther than buckets with a small distance between them. Lastly, we increment the distance between any two consecutive buckets by one to avoid dealing with distances of zero for later calculations.

Afterwards, we calculate the total distance $\mathcal{TD}$ as the sum of all distances between consecutive buckets. Having the total distance and the latest starting time L, we can now map the distance between two consecutive buckets to a portion of the $[0, \mathrm{L}]$ interval. We further define $\mathcal{CD}(B)$ as the cumulative distance up to some bucket B in the bucket list. The cumulative distance for the first bucket in the list consequently is zero, while the cumulative distance for the last bucket of the list is equal to the total distance $\mathcal{TD}$. The starting time $\mathcal{S}(B)$ for all flows of some bucket B in the bucket list is then defined by

$$\mathcal{S}(B) = (\mathcal{CD}(B) \div \mathcal{TD}) * \mathrm{L}$$

All in all, we conclude that for balanced tree topologies we are guaranteed to find a valid order of flows, and thus can compute a bucket list fulfilling the described properties. For other topologies, we may need to resort to an approximation of the order due to cyclic relationships. Having found the desired order for a balanced tree topology, we intuitively should also end up with a feasible solution to the corresponding No-wait Packet Scheduling Problem, if we distribute the buckets over the $[0, \mathrm{L}]$ interval. This holds true up to some threshold of flows that are to be scheduled. The more flows we are ordering with our algorithm, the larger the number of buckets in our bucket list can potentially grow. Assume the worst-case scenario, where we are scheduling n flows with the relation chain $f_1 \equiv f_2 \equiv \cdots \equiv f_n$. Here, we end up with exactly n buckets that each contain exactly one flow. Consequently, the larger n grows, the smaller the interval between two consecutive buckets can become. If n reaches the point, where the interval between two buckets is so small that their flows collide with each other, then we cannot obtain a feasible solution to the NW-PSP anymore.

The threshold up to which a feasible solution to the NW-PSP can be obtained is thus influenced by the number of flows to be scheduled and the type of their pairwise relations, as well as by the length of the cycle. For a very long cycle, we can distribute more buckets without introducing any collisions, while for a rather small cycle length the maximum number of buckets for which a feasible solution can be obtained also decreases.

6 Evaluation

We will now present the results obtained through a thorough evaluation of the presented desynchronization algorithms for IEEE Time-Sensitive Networks. We begin by discussing the scope of this evaluation and provide an overview over the structure of the remainder of this chapter. We will introduce the different network topologies used for benchmarking, as well as the set of generated problem instances. When presenting our network topologies, we will also use this opportunity to list the set of assumptions made for this evaluation, especially regarding the magnitude of the various network delays. To enable a better interpretation of the desynchronization results and their quality, we further introduce the two desynchronization metrics *network desynchronization score* and *schedule desynchronization score*, which we later try to correlate with the obtained results.

6.1 Evaluation Setup

With respect to network topologies we focus on tree topologies, which is consistent with our system model targeting Ethernet networks where switches are typically embedded into a spanning tree. To evaluate the performance of the presented desynchronization algorithms, we use trees of different sizes, namely small and large topologies. In Chapter 5 we have further observed that balancing properties might impact the performance of some desynchronization algorithms. Therefore, we consider balanced and imbalanced tree topologies. In detail, we present results for the following four topologies in the remainder of this chapter.

- A small balanced tree topology consisting of 19 nodes, of which 12 are host nodes
- A large balanced tree topology consisting of 120 nodes, of which 81 are host nodes
- A small imbalanced tree topology consisting of 22 nodes, of which 12 are hosts nodes
- A large imbalanced tree topology consisting of 123 nodes, of which 80 are host nodes

Although our focus is on tree topologies, we also performed preliminary evaluations on two grid topologies. These results are not presented here in detail, as our observations suggest that the performance on grid topologies closely resembles the results for imbalanced trees.

The small topologies correspond exactly to the two topologies presented in Chapter 4. The large topologies are scaled up versions of their small counterpart, where we increased the height and the fan-out. However, they retain the same characteristics as the respective small topology of the same type.

A major goal for the selection of the topologies was to provide a roughly similar difficulty to the scheduling problem for topologies of the same size. Furthermore, the large increase in network nodes for both large tree topologies is less noticeable, as all incoming and outgoing links of the tree root will remain bottleneck links and thus limit the maximum traffic that can be scheduled within a cycle period.

For imbalanced tree topologies, the factor most important to us was to have the host nodes distributed across as many levels of the tree as possible. As a result, we end up with a wide range of different path lengths between source hosts and destination hosts, which will maximize the differences of balancing properties between the two evaluated topology types. Since we cannot guarantee that the scheduling complexity for all topologies is completely equivalent, we cannot directly compare the results of one desynchronization algorithm between different topologies. We can however compare the performance of a desynchronization algorithm with respect to some other algorithm for a single topology. The difference in performance can then also be compared over different topologies. For example, we might see that some approach *A* performed better than approach *B* on one topology, but approach *B* performed better than *A* on another topology.

We assume every link of all topologies to operate with a link speed of 100 Mbps (Fast Ethernet). We further assume the frames of every flow to be 600 B large, which is in the middle between the typical minimum and maximum Ethernet frame size of 64 and 1500 Byte, respectively. This results in a transmission delay of 48 µs. At a propagation speed of $\frac{2}{3}$ lightspeed [DK+14], which is a typical value for fibre-optical cables, the propagation delay is 22 ns for cables with a length of 4.4 m. The required idle period after transmission of a data frame, the *inter frame gap*, is required to be equal to the transmission time of 96 bits. For Fast Ethernet, this results in an idle period of 960 ns. The processing delay for all switches of the network is assumed to be 4 µs. Lastly, we set the cycle period of the flows and of the schedule to 10 ms.

For every topology, we created multiple test sets as input data for the desynchronization algorithms and the ILP solver. For the two large topologies, these test sets range from 100 flows to be scheduled to 800 flows. For the small topologies, the upper limit of flows is set to 600 instead. The flow number of the test sets increases in steps of 100, resulting in a total of eight test sets for large topologies and six test sets for small topologies. We argue that 800 flows is a suitable upper bound on the scheduling complexity, as increasing the flow number any further has shown to often yield infeasible problem instances for the majority of our topologies.

Each test set is composed of ten different scenarios. The number of flows of each scenario belonging to the same test set is uniform, however, the source and destination hosts of each flow vary between the scenarios, as they are determined uniformly at random. We executed ten different scenarios instead of repeatedly executing the same scenario ten times to reduce the impact of unusual hard scenarios on the results. Furthermore, some scenarios may favour certain desynchronization approaches. In other words, by executing ten different scenarios with the same number of flows n, we approximate the distribution over all possible scenarios generated with n flows. Having results to ten scenarios per test set further enables us to calculate a 95% confidence interval for the median according to [Bou11]. Obviously, executing more scenarios would increase the confidence and better approximate the mentioned distribution. However, the choice of ten scenarios per test set is a trade-off between the required time for our experiments, which is limited, and the confidence.

Before conducting an evaluation on the full variety of generated test sets, we first evaluated a small sample of scenarios for different topologies and flow numbers. As we have presented several iterative desynchronization algorithms in Chapter 5, this preliminary evaluation aims at finetuning the termination criteria for these algorithms, to give them a reasonable chance to converge, but also terminate quickly, as this one of the demands for desynchronization algorithms used in a fast preprocessing stage before feeding the generated values to an ILP solver. Here we examine the change of two desynchronization scores, introduced later in this chapter, with increasing number of iterations, as well as the rate of convergence. With these gathered insights we then determine suitable upper bounds per iterative desynchronization algorithm, which are used for any further evaluations.

The remainder of this chapter is then split into two sections. The algorithms evaluated are the following:

- Naive Desynchronization (ND, Section 5.1)
- Relative Desynchronization (RD, Section 5.2)
- Link Desynchronization (LD, Section 5.3)
- Extended Link Desynchronization (ELD, Section 5.4)
- Ordered Desynchronization (OD, Section 5.5)
- Random phase initialization (random)

Here, the random approach simply initializes all phases of the flows uniformly at random and serves as a baseline throughout this evaluation. We want to clarify that random initialization of phases can by no means be viewed as a worst-case scenario in terms of desynchronization. While neither network topologies nor the relationships between flows are considered, assigning phases randomly will at least result in a uniform distribution of phases over the interval.

In the first section we conducted a standalone evaluation of algorithms listed above. We executed all algorithms on the generated test sets of every topology. We will then evaluate the schedule obtained through the desynchronization process with respect to the network desynchronization score and the schedule desynchronization score, which will be introduced in the following section. Furthermore, we examine the runtime of the desynchronization algorithms as well as the number of collisions in the resulting schedule.

For the second section of this evaluation, we then supply the results of the approaches listed above as input to the Gurobi ILP solver. Here we emphasize again that the results obtained through desynchronization will often be infeasible solutions to the scheduling problem, only serving as hints to guide the ILP solver. To measure the benefits of providing an initial set of hints, we additionally execute Gurobi on the generated test sets without providing any hints at all (*no hints*). The performance metrics used for this section of the evaluation are the runtime of Gurobi until the first feasible solution to the scheduling problem has been computed (*time to first solution*, TTFS), as well as the proven optimality bounds of found solutions.

All results were obtained by extracting them from a *log file* generated by the Gurobi solver during runtime. It specifies the timestamp for each found solution along with its respective optimality bound. As the timeframe for our evaluation was limited and we were evaluating many different

scenarios, we have set a time limit of two hours for any execution of the Gurobi ILP solver to avoid excessive runtimes. Once this time limit is reached, Gurobi will terminate. The log file will then contain all solutions found until reaching the time limit.

While in the first section the execution of the desynchronization algorithms itself is not heavily impacted by the complexity of the scheduling problem, we selected only a subset of the available test sets to evaluate in the second section. Here we picked test sets with a complexity high enough that the quality of provided hints may reflect on the results. However, we did not pick test sets with a complexity that high that we are not able to calculate any feasible solutions within the given time limit. Thus, we limited the evaluated number of flows for the two small topologies to 500. For the larger topologies we evaluated test sets with up to 600 flows. For any larger test sets, preliminary executions have shown that the number of results obtained within the time limit is too small for statistical evaluation.

All evaluations have been conducted on a dual socket machine with two AMD EPYC 7451 24-core processors with two threads per core and a total of 512 GB of memory. The operating system was Ubuntu 18.04.5 LTS and the utilized version of the Gurobi solver was 8.1.1. Due to the large number of executions and their respective runtime, we partially executed the different desynchronization algorithms for both sections of the evaluation in parallel. By pinning the processes of each experiment to the cores of an individual NUMA node we isolated different executions to avoid interference between concurrent experiments. Each *NUMA* node was comprised of six cores. We have seen Gurobi utilize all six cores of a *NUMA* node, while the desynchronization algorithms themselves only utilized a single core.

6.2 Desynchronization Scores

In this section, we will introduce the metrics developed to evaluate the degree of desynchronization of a derived schedule. These metrics will be used in the following sections to compare the desynchronization approaches.

To allow for a comparison we transform the output of each desynchronization approach to the representation used by our Extended Link Desynchronization algorithm (Section 5.4). In this representation, we can access and evaluate the transmission times of all flows on all links of the network. Thus, we do not only consider starting times of a flow, but its sending times on each network link along its flow path.

Let $\mathcal{F}_l$ be the set of flows that traverse a link l of the network. We define the *link desynchronization score* s_l^{link} of this link l as

$$s_l^{link} = \sum_{f \in \mathcal{F}_l} \min_{f' \in \mathcal{F}_l \setminus f} |\Delta(\phi_{f,l}, \phi_{f',l})|$$

This is the sum of the minimum distances of all phases $\phi_{f,l}$ to any other phase $\phi_{f',l}$ on link l. We observe that the maximum possible link desynchronization score is 1, as the sum of distances to the closest neighbouring phase can never exceed the interval size. In fact, a link score of 1 indicates a perfect desynchronization of a link, where n phases on this link are spread out evenly over the whole interval with a gap of $\frac{1}{n}$. A link score of 0 indicates that the phases on the respective

link are all synchronized, and the corresponding flows are thus scheduled to be sent over link l simultaneously. If a link of the network is not traversed by any flows, then we do not calculate a link desynchronization score for it.

For the following definition we denote the set of network links that are traversed by at least one flow as $\mathcal{L}$. We now define the *network desynchronization score* s^{net} as the average of the link desynchronization scores of all network links $\mathcal{L}$.

Definition 6.2.1 (Network desynchronization score)**.**

$$s^{net} = (\sum_{l \in \mathcal{L}} s_l^{link}) \div |\mathcal{L}|$$

With the same reasoning as above, the network desynchronization score s^{net} cannot exceed the value of 1, where 1 indicates an overall perfect desynchronization of every link $l \in \mathcal{L}$. A network desynchronization score of 1 however is not very realistic, as the phase distance between two neighbouring flows on very busy links will be very low. In contrast, on links with very little traffic the distance between phases should be rather large instead. If now two flows conflict on a very busy link and on a less busy link, then the desynchronization on the busy link might not allow for a large phase distance on the less busy link, as the distance of two flows remains constant across their conflict links, due to shortest path routing, the properties of tree topologies, and uniform network delays.

As the network desynchronization score looks at the desynchronization from the perspective of the network, we further introduce the *schedule desynchronization score* s^{sched} as an alternative metric to measure the desynchronization from the perspective of the network flows.

To compute the schedule desynchronization score s^{sched} we first compute the *flow desynchronization score* s_f^{flow} for every flow f of the network. For this, we iterate over all network links the flow f traverses along its flow path. Let this set of links be denoted by $\mathcal{L}_f$. On every link $l \in \mathcal{L}_f$, we determine the phase distance $\Delta(\phi_{f,l}, \phi_{s,l})$ to its direct successor s on this link. We only consider the phase distance to the successor, as the distance from its predecessor p to f is implicitly considered when calculating the flow desynchronization score for flow p. After calculating the phase distance to the successor on every link on f's flow path, we take the minimum of all these distances since this minimum distance deciding whether f collides with any succeeding flow.

While the minimum distance gives a good indication on whether the flow collides with any flow sent afterwards, it provides little information about the quality of the desynchronization, as the distance to a successor s on some link l is heavily impacted by the number flows traversing this link. Assume that the minimum distance of f to any succeeding flow on any link of f's flow path is 0.2. If at no link l on f's flow path there is more than one other flow also traversing l, then we see that 0.2 indicates a rather poor desynchronization. If there is some link l on f's flow path which is traversed by four other flows, then a minimum distance of 0.2 indicates a perfect desynchronization from the perspective of flow f.

Consequently, we normalize the minimum distance we previously calculated with the maximum number of flows on any link $l \in \mathcal{L}_f$ to receive the flow desynchronization score s_f^{flow}. If a flow f is completely independent of every other network flow, we consequently cannot find a successor on any link. In this case we do not calculate a flow score s_f^{flow} for this flow.

For the following definition we denote the set of flows that conflicts with at least one other flow of the network as $\mathcal{F}$. Given the flow desynchronization score for every flow $f \in \mathcal{F}$, we compute the schedule desynchronization score s^{sched} by averaging over all flow desynchronization scores.

Definition 6.2.2 (Schedule desynchronization score)**.**

$$s^{sched} = (\sum_{f \in \mathcal{F}} s_f^{flow}) \div |\mathcal{F}|$$

We see that the schedule desynchronization score punishes a bad desynchronization on some link much more than the network desynchronization score, as it only considers the distance to a successor on the bottleneck link, while other possibly perfect desynchronized links are ignored.

6.3 Tuning Termination Criteria

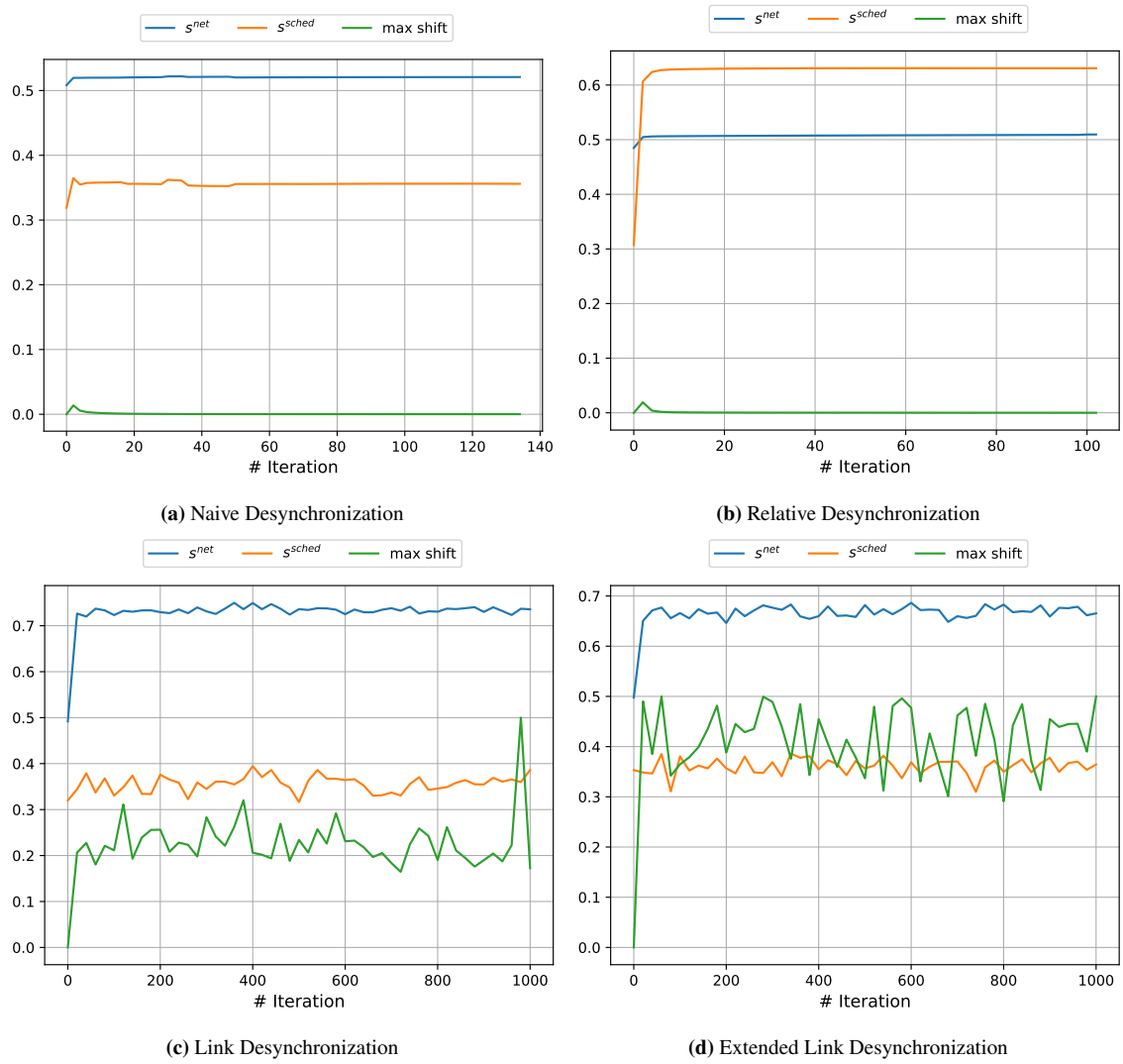
As convergence cannot be guaranteed for four of the presented algorithms, we first want to determine suitable upper bounds to their iteration number before conduction an extensive evaluation. The relevant approaches are the Naive Desynchronization, Relative Desynchronization, Link Desynchronization and Extended Link Desynchronization algorithms. To determine reasonable upper bounds per approach, we conducted a preliminary evaluation with a smaller set of scenarios to obtain a basic understanding on how the approaches behave with increasing numbers of iterations. Here, we set the maximum number of iterations to 1000 for all four approaches. Subject of our examination is the change in maximum phase shift per iteration, as well as the change of the network desynchronization score s^{net} and the schedule desynchronization score s^{sched} introduced in the previous section. Ideally, the maximum phase shift will decrease with each consecutive iteration so that the algorithm eventually converges into a desynchronized state.

Figure 6.1 shows results for an execution on a scenario containing 400 flows on the large imbalanced tree topology for the four algorithms under examination. The plots in Figure 6.1 only represent a single execution on a single scenario, although we executed each algorithm on several scenarios for this preliminary evaluation. However, we observed that while the actual values often differ between scenarios, the overall patterns remained the same.

The results showed a very similar behaviour for Naive and Relative Desynchronization. Similarly, for Extended Link Desynchronization the results closely resembled the results of the Link Desynchronization algorithm.

The first thing we observe for all algorithms is an initial increase in either the schedule desynchronization score s^{sched} or the network desynchronization score s^{net}, depending on the objective the respective algorithm tries to maximize. After this initial increase the scores remain constant for Relative Desynchronization and Naive Desynchronization. For Link Desynchronization and Extended Link Desynchronization we observe oscillations for both scores, however, the general trend remains constant as well. The most notable difference between the four desynchronization algorithms is the evolution of the maximum phase shift over the iterations.

The maximum phase shift generally remained very low for Relative and Naive Desynchronization. For these algorithms we see that statistics are only visualized up to iteration 100 and 140, respectively, although the upper limit of iterations was set to 1000 for both algorithms. Here, the



(a) Naive Desynchronization

(b) Relative Desynchronization

(c) Link Desynchronization

(d) Extended Link Desynchronization

Figure 6.1: Evolution of the maximum phase shift, the network desynchronization score, and the schedule desynchronization score over the executed iterations.

desynchronization process for Naive and Relative Desynchronization converged for the presented scenario. The desynchronization converged because the maximum phase shift eventually fell below a predefined threshold. This threshold has been set to 0.0001. The same behaviour was recorded for all other scenarios executed with Naive or Relative Desynchronization in this preliminary evaluation.

For Link Desynchronization and Extended Link Desynchronization, we notice that the maximum phase shift was visibly higher initially and did not decrease over the course of the execution. While also oscillating heavily, it generally remained constant with increasing numbers of iterations. As a result, both algorithms based on Link Desynchronization did not converge on the given scenario and the maximum number of 1000 iterations have been executed. The same trend was recorded for the remaining scenarios of this preliminary evaluation.

Consequently, we argue that the iteration limit of 1000 is suitable for the Relative and Naive Desynchronization approaches. These first examinations indicate that often both algorithms will terminate well in advance, thus a limit of 1000 will not negatively impact the runtime. If Naive or Relative Desynchronization do not converge as fast on some scenario, we further argue that having a high limit for maximum iterations will have a positive effect on the quality of the solution, as we see the maximum phase shift steadily decrease by a very small margin with increasing number of iterations.

For Link Desynchronization and Extended Link Desynchronization we feel that it would put the algorithms at a disadvantage compared to our other approaches if we set the maximum limit of iterations to 1000 as well. As in both approaches the optimum regarding both desynchronization scores is usually found within very few iterations, but scores stagnate from this point onward without converging, the total runtime would not reflect the actual time the algorithm needs to compute a reasonably good solution. Additionally, we do not see any improvements towards the maximum phase shift. We thus pick 200 iterations as an upper bound for the approaches based on Link Desynchronization. Based on the results gathered in our preliminary evaluation, this upper bound still is large enough such that the initial increase in scores is completely included but avoids the execution of unnecessary iterations in which no progress is made at all.

The second termination criterion is the threshold for the maximum phase shift of an iteration. If this shift falls below the chosen threshold in any iteration, the algorithm terminates. As briefly mentioned before, this threshold has been set to 0.0001 for all iterative approaches and will remain at this value for the following extensive evaluation.

6.4 Standalone Evaluation of Desynchronization Algorithms

In the following section we discuss the results obtained by benchmarking the five presented desynchronization approaches as standalone algorithms. Here we are using the calculated start times for each flow directly and do not feed them as hints to an ILP solver. The presented data thus corresponds to the first phase of our proposed two-split of the solving process only. Obviously, such a standalone operation is most useful if desynchronization algorithms already output a feasible solution without any collisions. Although producing feasible solutions was not the primary goal of the desynchronization algorithms, which should only approximate such a feasible solution, it is still interesting to see whether these algorithms could actually be used as simple and efficient standalone solution for TSN scheduling. Here we evaluated runtime, the previously introduced desynchronization scores and the number of collisions, as the latter serves as a very intuitive metric for the quality of a TDMA schedule. Moreover, the runtime of desynchronization algorithms is not only relevant for this standalone evaluation, but also when used in a preprocessing step for the generation of hints for the ILP. In this case, the overhead is required to be small compared to the runtime of the ILP solver to increase the efficiency of the overall solution.

6.4.1 Runtime and Convergence

As the eventual goal of this thesis is to speed up the solving process of the Gurobi ILP solver by supplying initial hints generated through fast desynchronization algorithms, we will begin by evaluating the runtime of our five presented approaches. We exclude the random approach listed in Section 6.1 as we are not interested in the runtime of this baseline algorithm. More so, a simple random phase initialization will be executed almost instantly.

As for the iterative approaches, runtime is also closely related to the number of executed iterations. Thus, we will additionally examine the convergence behaviour of these algorithms.

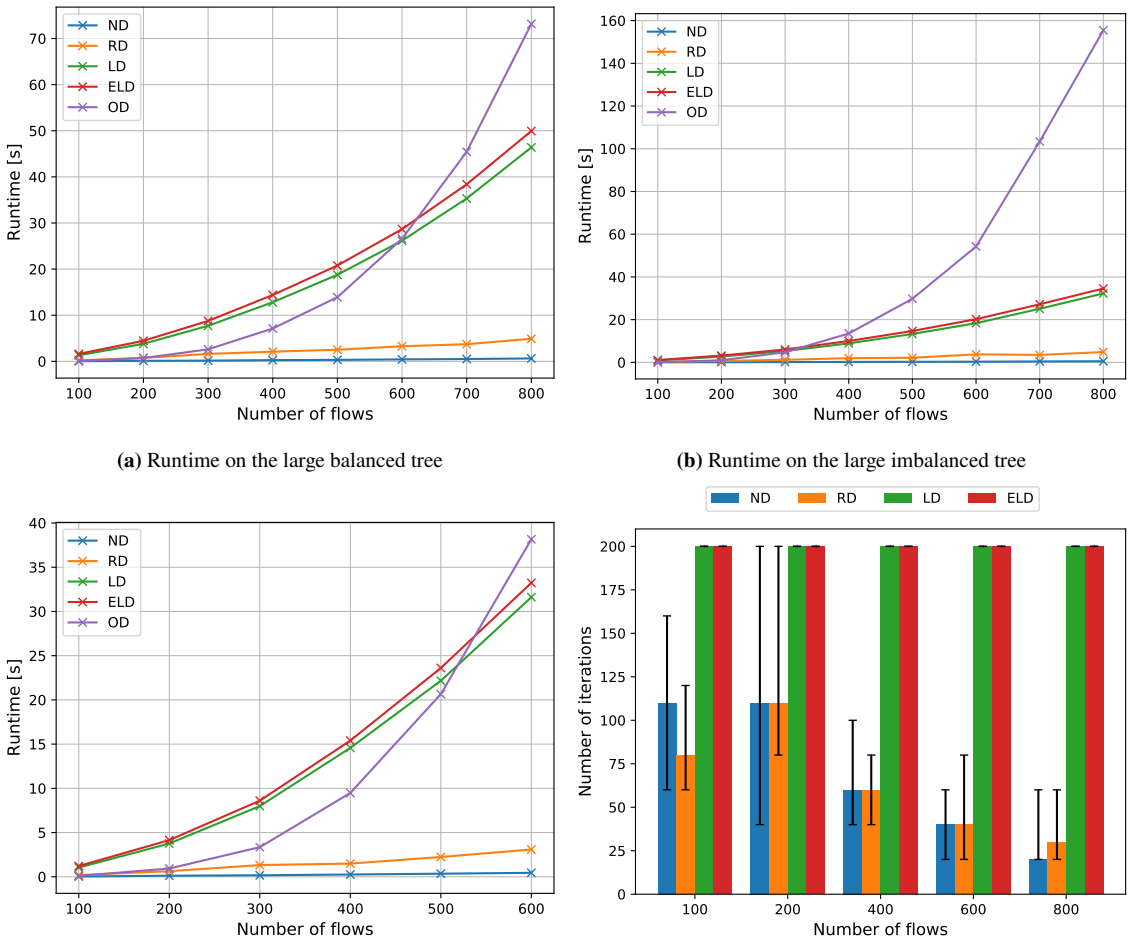
For measuring the runtime of our algorithms, the code has been stripped of all non-vital functionality. This includes all portions of code that gather statistics for any other metric than the runtime itself. Runtime of an execution is defined as the sum of the system and user time of the program and recorded with nanosecond precision. For Naive Desynchronization, Relative Desynchronization, Link Desynchronization and Extended Link Desynchronization the measurement begins when the phases are initialized and ends after the desynchronization process terminates, either by reaching the maximum number of iterations or by converging. For Ordered Desynchronization, measurement begins before the first bucket is initialized and filled with flows and ends after all sorted buckets have been distributed across the schedules cycle. As a result, the measurement includes all computations made during desynchronization but excludes all input and output operations.

All runtime results discussed in this section correspond to the median runtime of the execution of all ten scenarios for a specific test set. For better visual clarity of our presented graphics, we will refer to the different desynchronization algorithms by their acronyms introduced in Section 6.1 in any plot presented in the remainder of this thesis.

Figure 6.2a now shows the median runtime for all five desynchronization algorithms on the large balanced tree topology, while Figure 6.2b shows the median runtime on the large imbalanced tree topology.

On both topologies we see that all five algorithms terminated almost instantly for very small test sets with 100 flows. With increasing complexity of the problem instances, we observe that the runtime of Naive and Relative Desynchronization remained very low. Especially for Naive Desynchronization, the runtime was constant and never exceeded one second regardless of the size of the problem instance. In comparison, Relative Desynchronization showed a very slight linear increase in runtime when scaling up the problem size. For the test set with 800 flows on the large balanced tree, the median runtime for Relative Desynchronization was seven seconds. This difference compared to Naive Desynchronization is due to the additional computation of relative phases φ during phase updates.

As expected, we further see that both approaches based on Link Desynchronization yielded very similar results. In contrast to the two previously discussed algorithms, we see a noticeable increase in runtime when scaling up the problem size. From 100 to 800 flows the median runtime already increased by almost 40 times for the large imbalanced tree, and even by a factor of almost 50 for the large balanced tree. We can attribute the difference in runtime between both Link Desynchronization approaches and Naive or Relative Desynchronization to two factors. On one hand, Naive and Relative Desynchronization update the phase of each flow exactly once per iteration. The Link Desynchronization approaches however assign multiple phases to each flow, updating each flow per iteration as often as the number of links traversed by this flow. On the other hand, we refer to



(a) Runtime on the large balanced tree

(b) Runtime on the large imbalanced tree

(c) Runtime on the small balanced tree

(d) Convergence on the large balanced tree

Figure 6.2: Median runtime and rate of convergence of the TSN desynchronization algorithms

the plot shown in Figure 6.2d. Here we see the median number of iterations until termination for all four iterative approaches over the size of the test sets. The data shown in the plot corresponds to the execution of ten scenarios per test set on the large balanced tree topology, however the convergence results did not vary between different topologies. The error margins of each bar show both the maximum and minimum number of iterations to convergence or termination out of the ten executions. Remember that the upper bound for the number of executed iterations for both Link Desynchronization approaches has been set to 200. In our evaluation, these approaches never converged before this upper bound was reached. Naive and Relative Desynchronization usually converged at around 100 iterations or below. Most importantly we observe that for Naive and Relative Desynchronization the number of iterations needed to converge did not increase when scaling up the problem instance. In fact, we even see a decrease of executed iterations until convergence with scaled problem instances. Generally, with a higher number of flows a higher number of updates are performed per iteration, but with the decreasing number of iterations this balances out and contributes to the almost constant runtime of both approaches. Link Desynchronization and Extended Link Desynchronization executed more iterations and perform more phase updates per iteration, resulting in a higher runtime, especially when scaling up the problem instance.

Lastly for Ordered Desynchronization, we can divide the algorithm into two main parts. The calculation of the order itself, and the determination of sending times for each bucket of the resulting bucket list. Here, calculation of the order contributes by far the most towards the runtime of this approach. For small test sets Ordered Desynchronization showed a low median runtime on every topology, nearly comparable to the runtime of Naive and Relative Desynchronization. Scaling up the problem size however, we observe a strong growth in runtime. During every step of the ordering process, we conduct a pairwise comparison between all flows in the current bucket. Scaling up the number of flows, both the maximum possible number of steps and the number of pairwise comparisons per step increases. Thus, for the large balanced tree topology at 800 flows, we already see that Ordered Desynchronization showed the highest median runtime at 70 seconds, which is a 50% increase compared to Link Desynchronization.

On the large imbalanced tree, the median runtime of Ordered Desynchronization scaled even worse compared to our other desynchronization algorithms. For the largest test set with 800 flows, this median runtime was four times as high as for either of the Link Desynchronization algorithms. As discussed in Section 5.5, for imbalanced tree topologies we cannot guarantee that the flows can be ordered correctly, as cyclic relationships cannot be avoided. The larger the problem instance gets, the higher the probability becomes that cyclic relationships exist. In the presence of cyclic relationships, we further outlined how the algorithm theoretically always would move all flows of this cyclic relationship up to the next bucket. We addressed this issue by leaving at least one flow in the lower bucket. Still, we see how moving all flows but a single one up to the next bucket can result in a large number of total buckets. As a result, the number of pairwise comparisons only reduces by a very small amount with each step. In a balanced tree topology, given the absence of cyclic relationships, we will run into scenarios a lot more often where a larger number of flows remain in the lower bucket in a single step. The number of remaining flows to be ordered thus decreases faster.

The variance in runtime over the ten scenarios of a test set was generally very low for all desynchronization algorithms and topologies. Even for test sets with a comparably large number of flows our results usually only diverged by a maximum of two seconds from the respective median. Consequently, while the number of flows of a scenario did have a huge impact on the runtime, the actual distribution of the flows across the network hosts did not. The only exception to this was Ordered Desynchronization on imbalanced tree topologies, as we outlined that the runtime is heavily influenced by the number of cyclic relationships, which may vary between different scenarios of the same test set. For the test set with 800 flows the runtimes of the ten scenarios ranged from 143 seconds up to 184 seconds on the large imbalanced tree topology.

For completeness, we further briefly examine the impact of the topology size on the median runtime. Figure 6.2c shows the median runtime for the small balanced tree topology. Here the evaluated test sets range from 100 to 600 flows only, as the scheduling complexity on smaller topologies is higher.

At first glance we already see that the depicted curves closely resemble the results of the large balanced tree topology. However, it is especially surprising to see the median runtime of both Link Desynchronization approaches closely match their runtimes measured on the large balanced tree topology. Intuitively, as they iterate over the network links, runtime should increase proportional to the number of links in the network. Nonetheless, the results indicate that with increasing network size the network flows are less likely to conflict with one another, as they are more spread out across the network. Thus, even though there are more links to iterate over, on each link there are less

flows traversing it. Consequently, while more network links are being desynchronized, the time complexity of each separate desynchronization decreases. For Ordered Desynchronization, the slight decrease in median runtime from the small to the large topology can also be attributed to the same reason. Having a larger network and thus having less conflicts between flows results in more pairwise independent flows. Thus, the number of flows in a single bucket also increases, and consequently the number of flows to compare in each step reduces faster. We omit to present results for the small imbalanced topology, as they show the same similarities and differences to the large counterpart as already observed for the balanced tree topologies.

To conclude this subsection and to enable a better interpretation of the presented results, we want to outline that the runtime of desynchronization algorithms is only influenced by the number of flows to be scheduled and the respective topology. Especially, it is not at all affected by the scheduling complexity of the problem. We see that given a set of flows to be scheduled on a given topology, the length of the schedule cycle defines the complexity of the scheduling problem. A large cycle simplifies the calculation of a feasible solution. During desynchronization, however, the schedule cycle is not considered at all. It is only taken into consideration when mapping the desynchronization results back to a concrete sending time. As a result, if the number of network flows grows extremely large, our results suggest that Link Desynchronization, Extended Link Desynchronization and Ordered Desynchronization possibly will not fulfil the demand for a fast heuristic to calculate hints for the NW-PSP anymore.

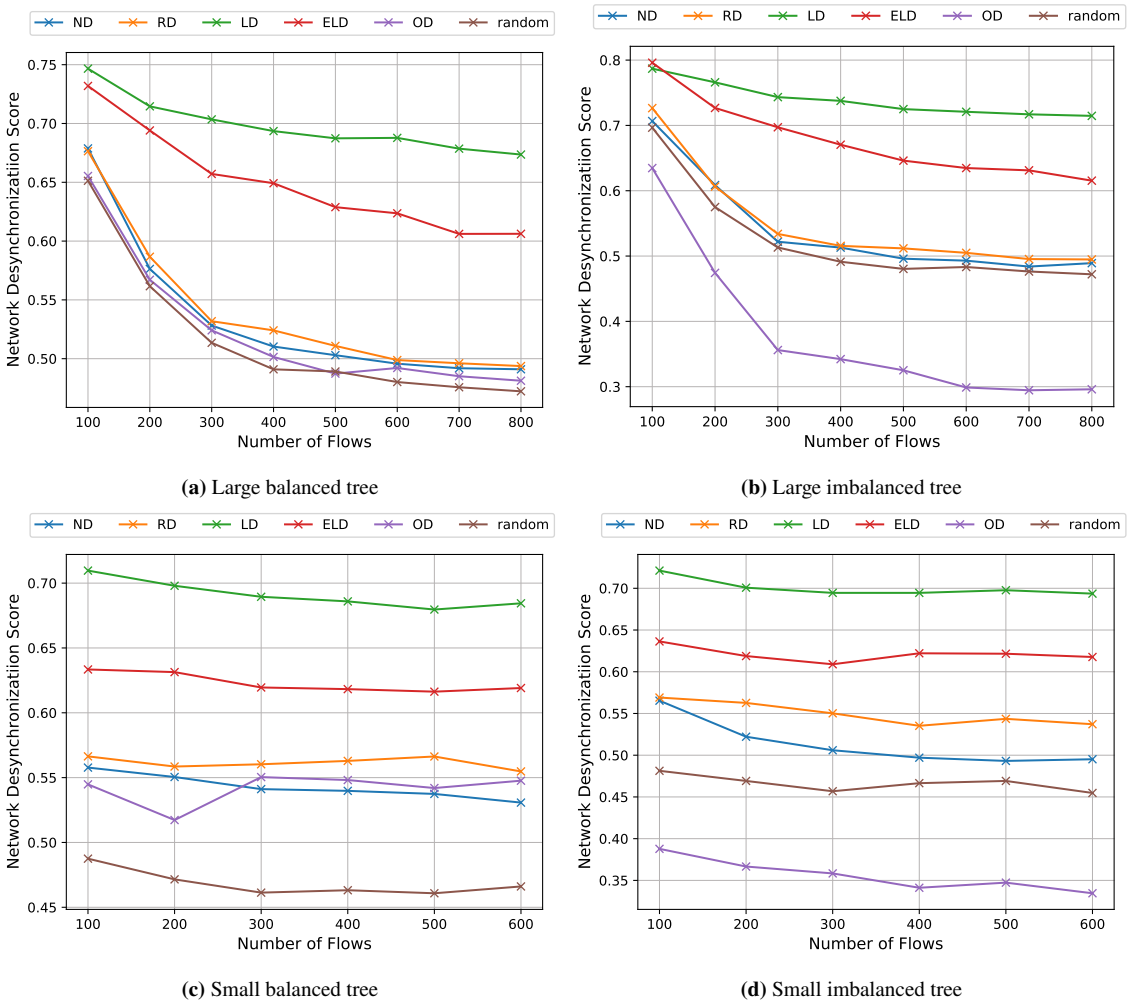
6.4.2 Network and Schedule Desynchronization Score

After having discussed runtime and convergence, we will now move on towards examining the solution quality of our desynchronization algorithms with respect to the previously defined desynchronization scores. Currently, it is still very hard to determine a single metric that best measures the quality of a desynchronization. Consequently, we do not aim to establish a total ranking of our approaches, but rather try to provide an intuition on their strengths and weaknesses. By doing so, we hope to detect correlations to these results in the sections to follow.

Figure 6.3 shows the median network desynchronization score s^{net} over the test sets for the large balanced tree topology and the large imbalanced tree topology. For both topologies, the two Link Desynchronization algorithms yielded by far the highest scores. This was expected behaviour, as the network desynchronization score corresponds closely to the objective that Link Desynchronization tries to optimize.

For small problem instances the network desynchronization score was comparably high for all six evaluated approaches. When scaling up the problem instance, we see that the scores decrease, although for some desynchronization algorithms more rapid than for others. The farther we scaled the problem instance, the smaller this decrease becomes, and the network desynchronization score eventually converged against a certain value.

For larger problem instances, the median network desynchronization score for Link Desynchronization was slightly below 0.7, while Extended Link Desynchronization scored noticeably lower at around 0.6. We conclude that the negative aspects introduced with our Extended Link Desynchronization outweigh the desired positive effect targeted by the modifications made.



(a) Large balanced tree

(b) Large imbalanced tree

(c) Small balanced tree

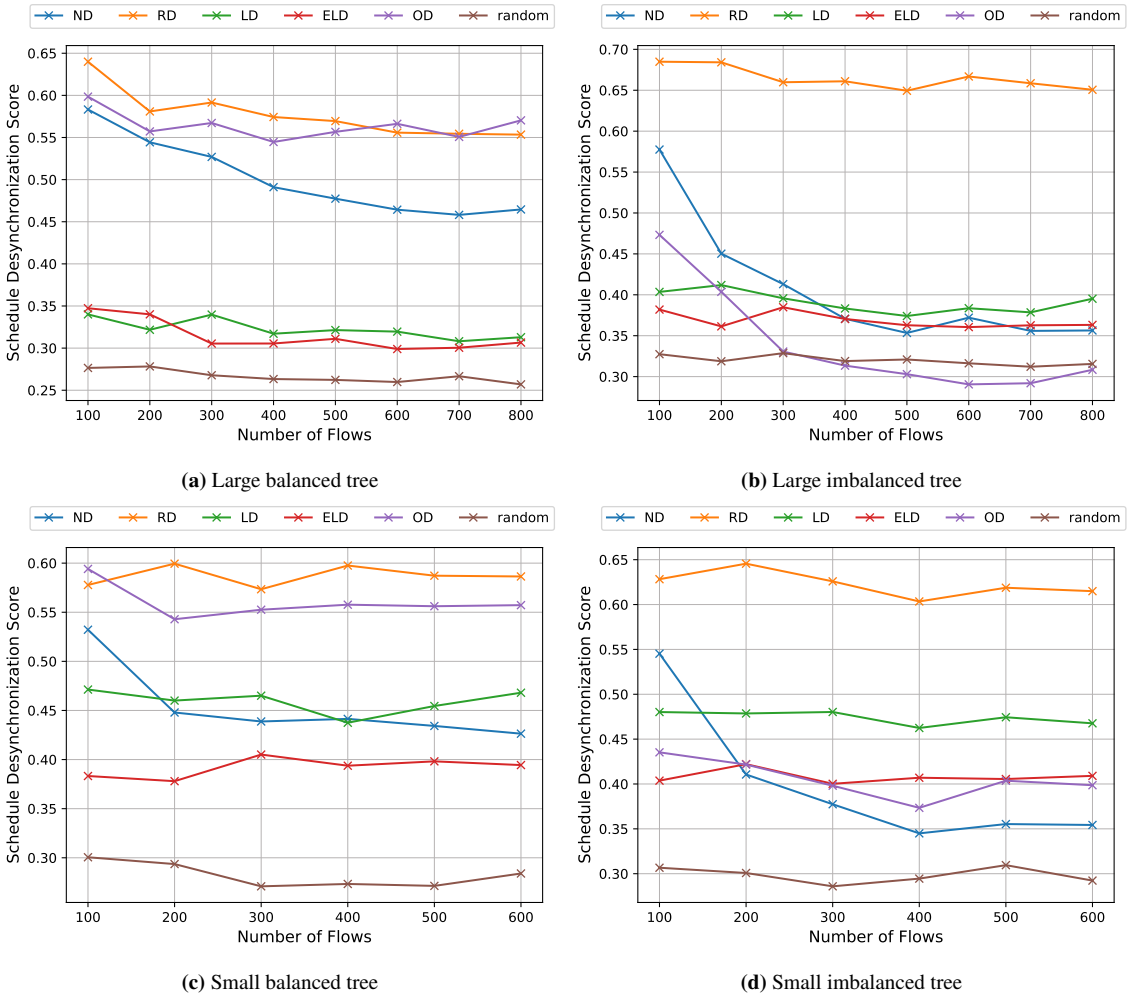
(d) Small imbalanced tree

Figure 6.3: Network desynchronization score

The three remaining desynchronization algorithms achieved very similar network desynchronization scores for the balanced tree topology with a median score of around 0.5. More so, their network desynchronization scores failed to show significant improvements over the score of the random approach.

In contrast, for the imbalanced tree topology we observe that the network desynchronization score for Ordered Desynchronization deteriorated very quickly, falling to 0.3 for large problem instances. The implications of imbalanced tree topologies for Ordered Desynchronization have been outlined multiple times throughout this thesis already.

For small topologies (cf. Figure 6.3c and Figure 6.3d) the gathered results closely resemble the results depicted in Figure 6.3a and Figure 6.3b with respect to the actual values of the network desynchronization score. However, it is notable that here we did not see the network desynchronization score start at a comparably high level for small instances and decrease when scaling up the problem instance. Instead, they initially achieved a lower network desynchronization



(a) Large balanced tree

(b) Large imbalanced tree

(c) Small balanced tree

(d) Small imbalanced tree

Figure 6.4: Schedule desynchronization score

score for small problem instances, but the decrease when scaling up the problem size was less pronounced. As the scheduling complexity for small topologies is generally higher, we assume that convergence against a certain value simply occurred earlier compared to large topologies.

Figure 6.4 now depicts the median schedule desynchronization score s^{sched} for the two large tree topologies. In contrast to the network desynchronization score, we now observe that Relative Desynchronization achieved high scores on both topologies. Again, this is expected behaviour, as the schedule desynchronization score closely corresponds to the optimization goal of Relative Desynchronization.

It is however interesting to see that Ordered Desynchronization scored very similar to Relative Desynchronization on the large balanced tree topology. Furthermore, while both Link Desynchronization algorithms resided in the lower regions as opposed to the network desynchronization score, we see that all desynchronization algorithms yield a higher schedule desynchronization score than the random approach. Only exception to this was the schedule desynchronization score for Ordered

Desynchronization on the large imbalanced topology. As already observed previously for the network desynchronization score, the schedule desynchronization score of Ordered Desynchronization fell off heavily on the imbalanced tree topology.

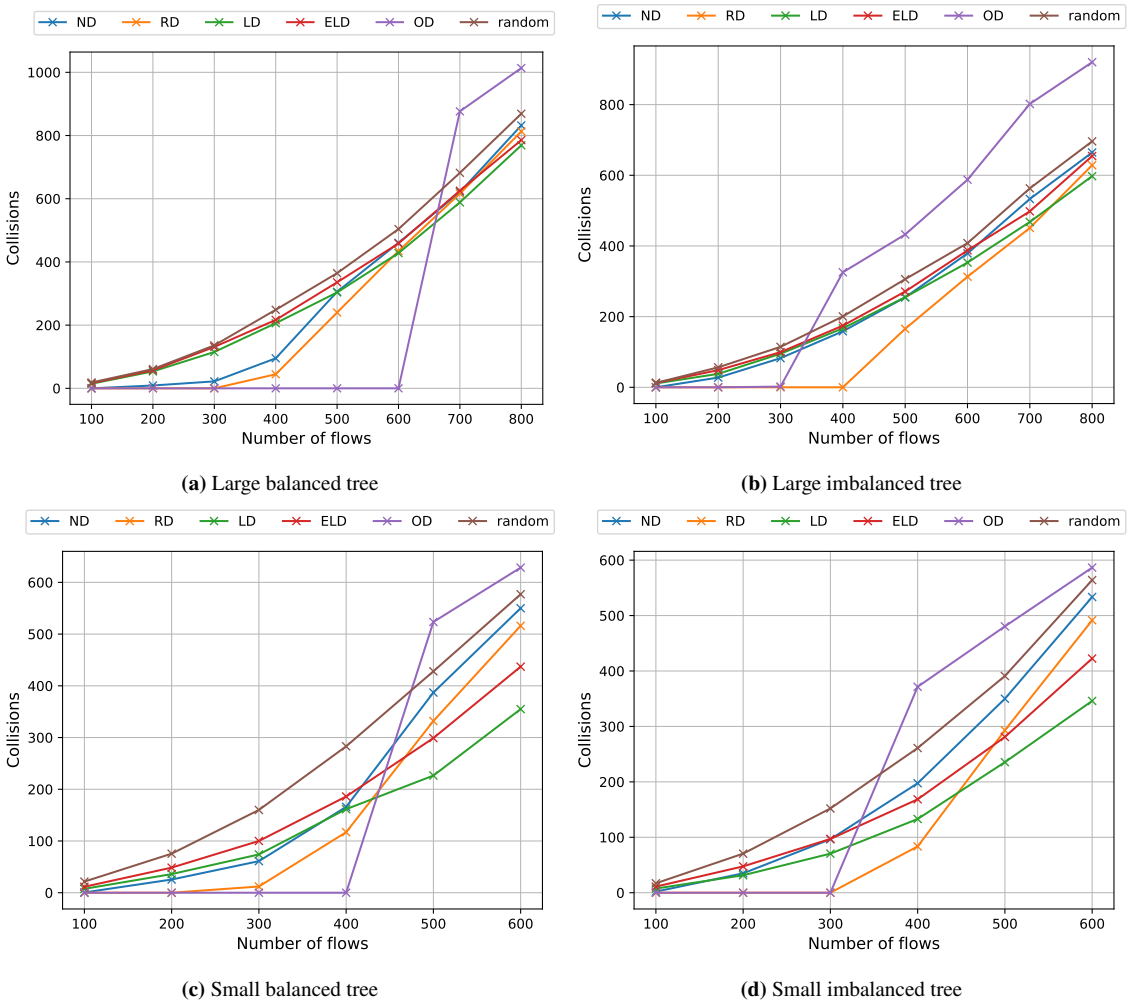
While Naive Desynchronization was able to achieve similar schedule desynchronization scores as both Ordered and Relative Desynchronization for the balanced tree, its score also deteriorated quickly with scaled up problem instances on the imbalanced tree topology. With a score of 0.65 on this topology, the schedule desynchronization score for Relative Desynchronization was already almost twice as high as the score of Naive Desynchronization for the largest test set consisting of 800 flows. The reason for this behaviour is the difference in path length of flows in both topologies. The set of flows to be scheduled is determined by picking source and destination hosts from the topology uniformly at random. If we assume that each sub-tree of the tree root node contains a similar amount of host nodes, a large portion of flows is expected to be routed over the root node. This is, because after picking a source host residing in some sub-tree, the probability that the chosen destination host will reside in another sub-tree should be at least 50% for trees with only two sub-trees, and even higher for trees with a larger fan out. In a balanced tree topology, where all host nodes reside on the same tree level, the path length of every flow traversing the root node is equal. As already discussed in Section 5.5, the path length up the conflict link for conflicting flows that both traverse the root node is equal as well. Having equal path length up the conflict link, we then observe that the relative phase $\varphi_{g,f}$ of g from the perspective of f will be the same as the phase ϕ_g of g. The improvements made from Naive Desynchronization to Relative Desynchronization consequently are only noticeable to a very small degree. In imbalanced trees, leaf nodes are not restricted to reside on the same tree level anymore. As a result, two flows traversing the tree root may also have different path lengths. Furthermore, having equal path length does not guarantee that the path length up to a conflict link is equal as well. Consequently, $\varphi_{g,f}$ will equal ϕ_g less frequently, emphasizing the advantage of Relative Desynchronization over Naive Desynchronization.

For small topologies (cf. Figure 6.4c and Figure 6.4d) we observe the same trend we observed for the network desynchronization score, where we see the values stay rather constant when scaling up the problem instance. Their actual schedule desynchronization score again was very slightly above the value against which it converged for large topologies.

All in all, each desynchronization algorithm scored high for one of the two evaluated desynchronization scores. No desynchronization algorithm however achieved both a high network desynchronization score and a high schedule desynchronization score. Further, the results of Naive and Ordered Desynchronization have been influenced heavily by the topology type, while Relative Desynchronization and both Link Desynchronization approaches performed consistent across all topologies.

6.4.3 Collisions

Lastly, we examine the number of collisions occurring in the schedules calculated by the desynchronization algorithms. The number of collisions is probably the most intuitive metric to interpret. The better the desynchronization, the less collisions should be present in the resulting schedule. The less collisions are present in the schedule, the less constraints of the corresponding No-wait Packet Scheduling Problem should be violated. Thus, the less collisions the desynchronization algorithms



(a) Large balanced tree

(b) Large imbalanced tree

(c) Small balanced tree

(d) Small imbalanced tree

Figure 6.5: Number of pairwise collisions after desynchronization

produce, the better the resulting schedule should approximate a feasible solution to the scheduling problem. However, it must be noted that a schedule with little to no collisions does not necessarily imply a good desynchronization.

Given the result of any desynchronization algorithm, we can trace the transmission times of a flow at any network link by utilizing the defined network delays. If at any network link the interval between the start of two distinct transmissions is smaller than the transmission delay plus the inter frame gap, both flows collide. For this evaluation, we recorded the total number of pairwise collisions. Thus, we only accounted for a collision between two flows f and g once, instead of recording it as one collision for f and a separate collision for g. Furthermore, if two flows collide on multiple links of the network, we still only recorded a single collision.

Figure 6.5a and Figure 6.5c show the median number of collisions for the desynchronization algorithms on the small and large balanced tree topology, while Figure 6.5b and Figure 6.5d show the results for the corresponding imbalanced tree topologies.

On both balanced tree topologies, Ordered Desynchronization has completely avoided collisions up to a certain threshold of flows. On the small balanced tree topology this threshold was exceeded at 500 flows, on the large balanced tree topology at 700 flows. As discussed in Section 5.5, this is since the number of different buckets the ordering created now was too large to distribute over the given cycle without causing any collisions. We see this breakpoint already exceeded at 500 flows on the small balanced tree topology, because here flows are more likely to conflict with each other. Having more pairwise conflicts the number of buckets consequently will be greater than in large topologies, where more flows are pairwise independent. For test sets larger than the given threshold the number of collisions for Ordered Desynchronization then increased drastically.

For imbalanced tree topologies we have shown in Section 5.5 that Ordered Desynchronization is not always able to correctly order the flows. As a result, it approximates such an order as closely as possible. We see that for small problem instances this approximation of an order often was sufficient to produce collision-free schedules, however we notice the steep increase in collisions at a much earlier point than we did for balanced tree topologies.

Relative Desynchronization yielded very little collisions for small problem instances and often was able to even compete with Ordered Desynchronization on balanced tree topologies. When scaling up the problem instances, we see that the number of collisions for both Relative and Naive Desynchronization increased at a higher rate than for the two Link Desynchronization approaches. These approaches performed mediocre on small instances, but for the largest test sets we then see that Link Desynchronization produced the least collisions out of all desynchronization algorithms. This held true across all topologies; however, the trend was best visible on the two small topologies. For the largest test set on small topologies, Link Desynchronization produced only 70% of the collisions of Relative Desynchronization, which provided the next best results after both Link Desynchronization algorithms.

Lastly, all desynchronization algorithms, excluding Ordered Desynchronization for imbalanced tree topologies and after exceeding its threshold, consistently produced schedules with less collisions than present in the schedule obtained through random phase initialisation.

6.4.4 Summary

As we now have evaluated our presented approaches as standalone algorithms, we will briefly summarize what we have learned throughout the last section before moving on towards examining the impact on the solving process of the Gurobi ILP solver.

We have seen that different approaches optimize different scores. Both approaches based on Link Desynchronization achieved high results for the network desynchronization score, while the other three approaches performed better with respect to the schedule desynchronization score.

We have further observed that no algorithm is consistently better than all other approaches across all topology types under examination. Correlating the results of both desynchronization scores with the number of collisions, we see that for small instances algorithms with higher schedule desynchronization scores tended to produce the least collisions. We further observed Ordered Desynchronization to completely avoid collisions for balanced tree topologies up to a given threshold. The produced solutions thus also are feasible solutions to the No-wait Packet Scheduling Problem. Consequently, for suitable topologies and a reasonable problem size, the Ordered Desynchronization

approach can even act as a standalone scheduling algorithm. However, we also observed that Ordered Desynchronization is not suitable for more complex topology types, as we cannot guarantee the calculation of the required order.

When scaling up the problem instances and thus trying to create denser schedules, we see that algorithms maximizing the schedule desynchronization score tended to scale worse regarding the number of collisions than the algorithms maximizing the network desynchronization score. This leads to the conclusion that for dense schedules the network desynchronization score is a more important indicator for desynchronization quality.

Further, we observed that Link Desynchronization and Extended Link Desynchronization as well as Relative Desynchronization are not noticeably impacted by the type of topology.

With the goal in mind to produce a fast intermediate solution, Link Desynchronization still comes with disadvantages compared to Naive and Relative Desynchronization. Even though the schedule quality may scale better for the former, this comes at the expense of a higher runtime. Still, the worst scaling in terms of runtime has been observed for Ordered Desynchronization. However, we argue that this is of little concern, as based on the presented results a usage of Ordered Desynchronization for very large problem instances cannot be recommended. For small and medium problem instances on balanced tree topologies, the longer runtime compared to Naive and Relative Desynchronization is compensated by the calculation of a feasible solution to the scheduling problem.

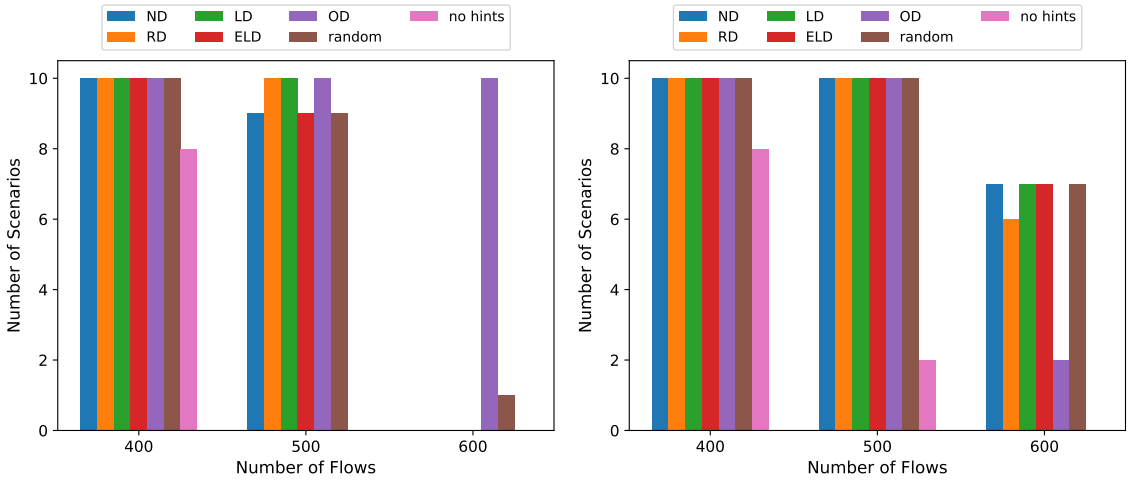
6.5 ILP Speed Up using Desynchronization Hints

After benchmarking the desynchronization approaches as standalone algorithms, we now inspect the impact on the Gurobi ILP solver when supplying desynchronization results as hints. Aspects under examination were the runtime of Gurobi until a first feasible solution to the scheduling problem has been found, the quality of this solution in comparison to an optimal solution, and how the quality of found solutions improves with increasing runtime. Due to the slightly lower complexity to the scheduling problem of our large topologies we were able to evaluate more test sets with a higher upper limit of flows. Consequently, we were able to gather more results than for smaller topologies. As a result, we will mainly present results for large topologies in this section, only including the small topologies if their results diverge noticeably. As a baseline for comparison, we additionally executed Gurobi on our test sets without provision of any hints at all. In graphics we refer to this execution by the term *no hints*.

6.5.1 Time to first solution

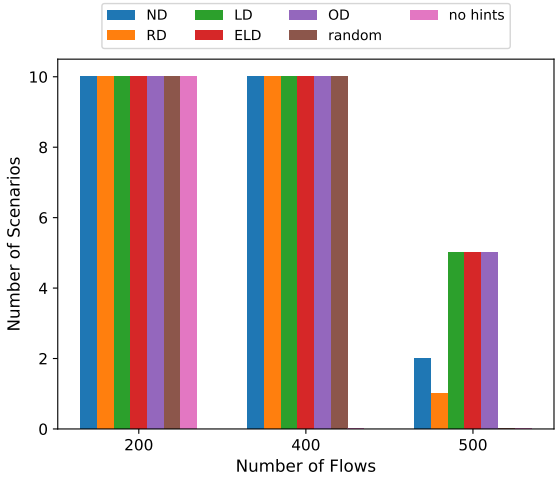
We begin by inspecting the time at which Gurobi found the first feasible solution to a given problem instance. This feasible solution fulfils all constraints of the No-wait Packet Scheduling Problem, but usually is not optimal as it does not minimize the makespan of the schedule. In the remainder of this thesis, we will also refer to this time taken to calculate a first feasible solution as the *time to first solution* (TTFS).

However, when evaluating the gathered results, we ran into some difficulties. To be able to execute many different scenarios for each desynchronization approach, we mentioned a time limit of two hours at which the Gurobi ILP solver terminates and returns the best solution found up to this point.

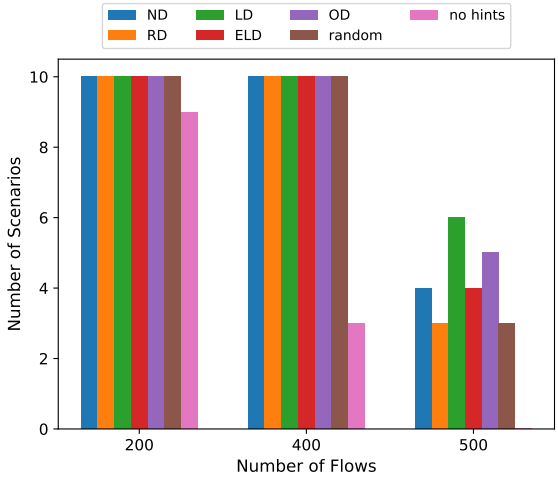


(a) Large balanced tree

(b) Large imbalanced tree



(c) Small balanced tree



(d) Small imbalanced tree

Figure 6.6: Number of scenarios per test set, for which at least one feasible solution to the ILP was found within the time limit (TTFS < 2 hours).

Even though we can guarantee that for every generated scenario there exists a feasible solution, we cannot guarantee that a feasible solution is always found within two hours. Especially when not supplying any hints to Gurobi we expect the TTFS to often exceed the set time limit.

To make the interpretation of the results presented in the remainder of this evaluation easier, we thus start off by inspecting the number of scenarios for which the Gurobi solver was able to determine a first feasible solution within the given time limit.

For hints generated by each desynchronization approach, as well as for executions without hints and with random hints, Figure 6.6 depicts the number of scenarios of a test set for which the Gurobi solver found at least one feasible solution before reaching the time limit.

For both large topologies we observe that for test sets with 400 flows almost for every scenario a feasible solution has been found no matter if hints were provided or not. For Gurobi executions with provided hints, this trend continued for the test set with 500 flows. In contrast to the smaller test set however, we see that feasible solutions were found for very little or none of the scenarios when not providing any hints. Lastly, for the test set of the large balanced tree with 600 flows, Gurobi was only able to consistently calculate a feasible solution within the time limit if the provided hints were generated by Ordered Desynchronization.

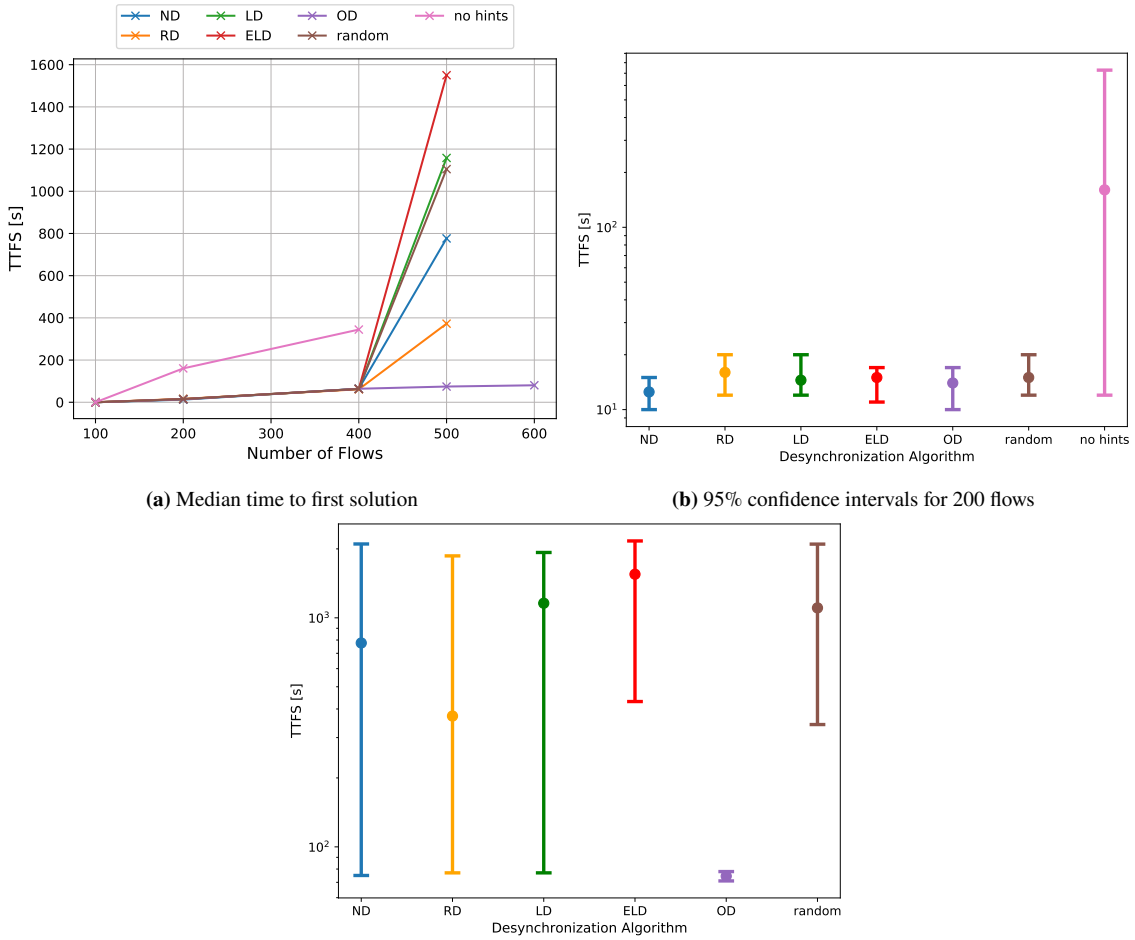
For the small topologies, we generally observe the same patterns. The breakpoint where Gurobi was not able to produce enough solutions without provision of hints was reached earlier than for large topologies. Additionally, given the set time limit, a reasonably large sample size of results for any approach could only be obtained for test sets with up to 400 flows due to the higher scheduling complexity.

We would like to emphasize that executions where no first feasible solution has been found can under no circumstances be ignored when analysing the results. However, the only guarantee we can give for these scenarios is that the TTFS was larger than the set time limit of two hours.

With this presented data in mind, we now move towards examining the TTFS of Gurobi. As we can guarantee the time limit to act as a lower bound for the actual time to the first solution, we see that we can always calculate the median TTFS, as long as for the majority of scenarios a first solution was found in less than two hours. This is because we can still guarantee that the actual TTFS for the missing scenarios is higher than for any of the scenarios to which a solution has been found. If for any approach, this might be the execution of Gurobi with hints generated by some desynchronization algorithm or the execution of Gurobi without hints, a feasible solution has been found within the time limit for six or more of the ten scenarios, we treat the runtime of every scenario for which no solution has been found as 7200 seconds, which again corresponds to the time limit. If there only have been found feasible solutions for five or less of the ten scenarios, we omit the data in the graphics to follow. Augmenting the incomplete set of data analogous to the approach described above is not applicable in this case, as we see that the number of scenarios missing a feasible solution is at least as large as the number of scenarios to which a solution has been found within the time limit. Assuming some value for the TTFS of the former, we see how the choice of this value has a direct influence on the median.

Computing the 95% confidence interval of the median is only possible if at least for nine of the ten scenarios of a test set a feasible solution has been found within the time limit [Bou11]. If at least for two scenarios a feasible solution has been found, we are only able to specify the lower bound of the corresponding confidence interval. If for no scenario of a test set a solution has been found within the time limit, the only guarantee we can give is that the lower bound of the 95% confidence interval of the median TTFS cannot lie below the time limit.

We begin by discussing the results for the large balanced tree topology. Figure 6.7a visualizes the median TTFS of the Gurobi solver with provided hints generated by our desynchronization algorithms, with hints generated through random phase initialization, and without any provided hints at all.



(a) Median time to first solution

(b) 95% confidence intervals for 200 flows

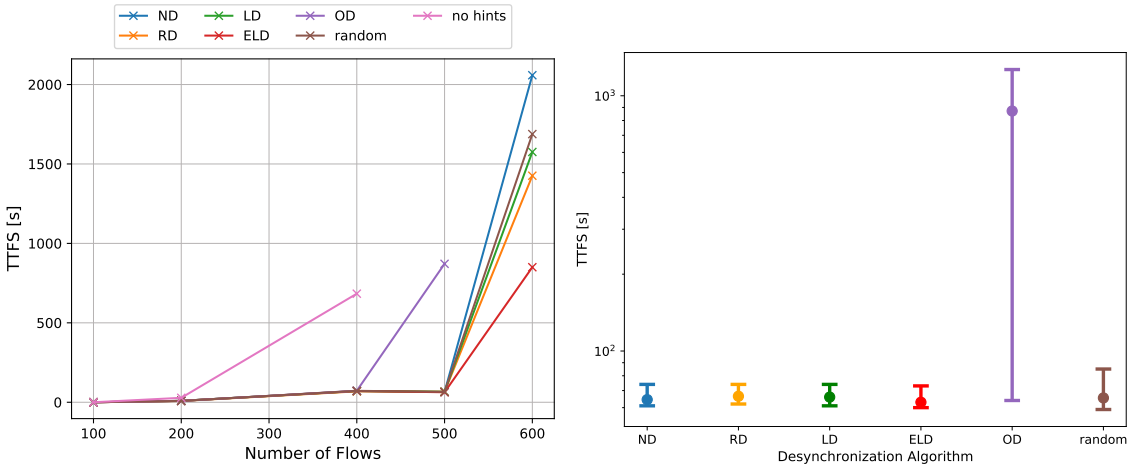
(c) 95% confidence intervals for 500 flows

Figure 6.7: Median time to first solution for the large balanced tree

For the smallest test set with 100 flows no difference between the provision of hints and the absence of hints can be observed. For test sets with 200 flows and more, we then already notice a difference in median TTFS between executions with provided hints and executions without any hints. For the test set with 200 flows specifically, the median TTFS without provided hints was already higher by a factor of ten.

Figure 6.7b further displays the 95% confidence intervals for median TTFS on the test set with 200 flows. Especially the much larger variance for executions without provided hints becomes apparent. As the confidence interval does still overlap with the hint-based approaches, we cannot guarantee that this difference is in fact significant, however, it seems very likely given the discrepancies for both medians and upper bounds of confidence intervals.

Furthermore, when providing hints for small problem instances up to 400 flows, the observed TTFS stayed very consistent regardless of the desynchronization algorithm used to generate hints. For the test set containing 500 flows, we then see the median times to the first solution diverge depending on the desynchronization algorithm used for hint generation. Especially the increase compared to



(a) Median time to first solution

(b) 95% confidence intervals for 500 flows

Figure 6.8: Median time to first solution for the large imbalanced Tree

the next smaller test set is very steep for most approaches. The median TTFS for executions without hints is now omitted for test sets with 500 flows and more, as no feasible solutions have been found within the time limit of 7200 seconds.

Based on Figure 6.7c, we can further guarantee that the median time to first solution without providing hints is higher by at least a factor of six compared to executions with supplied hints, as for no scenario a feasible solution has been found without hints within the time limit. Without provided hints, the lower bound of the 95% confidence interval thus was at least two hours, and consequently the same holds true for the median TTFS.

We further observe that the median TTFS with Ordered Desynchronization hints was lower by at least a factor of four compared to random hints or Extended Link Desynchronization hints. A difference to the remaining three desynchronization algorithms is very likely given the extremely small variance for Ordered Desynchronization, but the results do not allow for any guarantees on this. Excluding Ordered Desynchronization hints, we see that the median TTFS of Relative and Naive Desynchronization were the lowest amongst all remaining desynchronization algorithms. This suggests that the schedule desynchronization score better reflects the quality of a desynchronization with regard to the scheduling problem. According to Figure 6.7c however, the in Figure 6.7a observed differences in median TTFS between the remaining hint-based approaches for a test set of 500 flows cannot be considered significant.

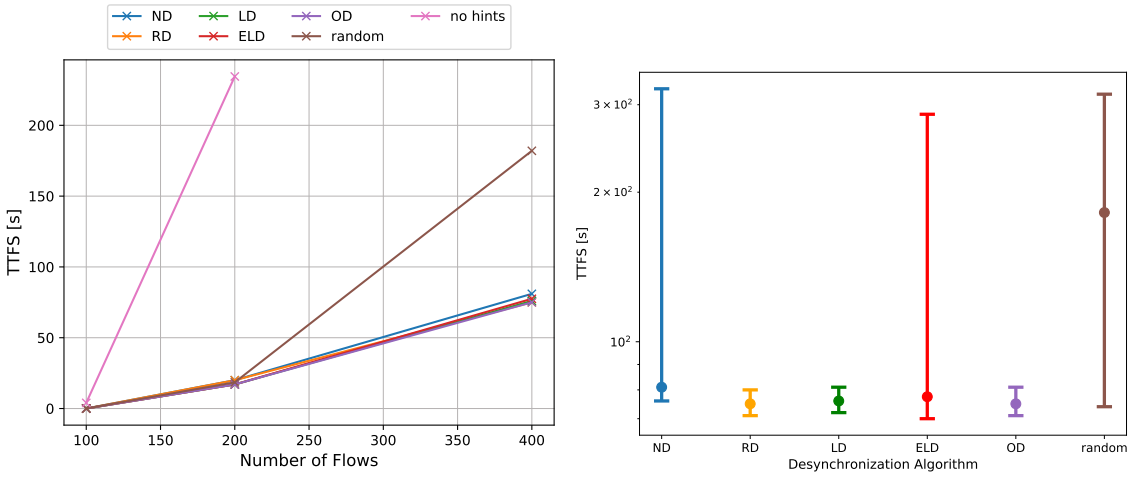
At 600 flows, solely the hints generated by Ordered Desynchronization enabled the Gurobi solver to compute feasible solutions within the time limit. It stands out that the median TTFS with hints generated by this algorithm remained nearly constant when scaling up the problem instance. We attribute this behaviour to the fact that Ordered Desynchronization was able to produce feasible solutions by itself to those problem instances. The TTFS of Gurobi was then reduced to verifying the supplied hints as feasible solution.

Moving on to the large imbalanced tree topology, for which the median times to the first solution are shown in Figure 6.8a, we observe the same speed up introduced by the provision of hints. On one hand we still see that the differences in quality of the provided hints had very little impact

on the median TTFS for smaller test sets, where all desynchronization algorithms yielded similar results. Very interesting on the other hand are the results gathered for the test set with 500 flows. While most desynchronization algorithms continued to produce similar results, we see that hints generated by Ordered Desynchronization increased the median TTFS by a factor of ten compared to all other hints. Previous evaluations have already shown that Ordered Desynchronization is not very well suited for topologies other than balanced trees and the produced desynchronization is comparably bad for them. What makes this behaviour extremely interesting is however the fact, that for the first time we observe that the quality of the desynchronization did have an influence on the TTFS. Previously, on the large balanced tree topology we have seen that the hints generated by Ordered Desynchronization led to a strong speed up. Here however, we had to attribute this speed up to the fact that the produced schedules of Ordered Desynchronization already formed a feasible solution to the scheduling problem, while the schedules of other approaches did not. As we learned in the previous section that for a test set with 500 flows on the large imbalanced tree no desynchronization algorithm yielded feasible solutions to the No-wait Packet Scheduling Problem, we cannot attribute this difference to the same reason anymore. Consequently, we now observe that a lower degree of desynchronization did lead to a much smaller speed up. The 95% confidence intervals shown in Figure 6.8b further show that the recorded difference in median TTFS is very likely to be significant.

For the test set with 600 flows, we then again see the median TTFS of different desynchronization algorithms diverge. Surprisingly, this time Extended Link Desynchronization recorded the lowest median TTFS out of all the evaluated approaches. Unfortunately, we cannot correlate or further support this observation with any results gathered during our standalone evaluation. Compared to results for the large balanced tree, we now see the TTFS with hints generated by Naive Desynchronization to be the highest among all desynchronization algorithms and even higher than for the random approach. The reasons for this drop-off have already been addressed in Section 6.4. Hints generated by Ordered Desynchronization did only enable Gurobi to find solutions to two of the ten scenarios within the time limit, and the results are thus omitted for this test set. However, as shown at the beginning of this section, we cannot present confidence intervals for this test set as the number of scenarios for which solutions have been found is too small across all algorithms. Significance of the observed differences for the largest test set thus cannot be guaranteed.

Lastly, we additionally briefly address our results gathered on the small balanced tree topology, as we observed a few interesting patterns there. Figure 6.9a shows the median TTFS for this particular topology. Compared to results for any other topology we do see a noticeable difference in median TTFS between the hints generated by our desynchronization algorithms and the randomly generated hints. At 400 flows, the median TTFS with randomly generated hints was with 180 s already more than twice as high. The corresponding 95% confidence intervals displayed in Figure 6.9b further indicate that this difference is likely to be significant compared to Relative Desynchronization, Link Desynchronization and Ordered Desynchronization. For the remaining two algorithms we cannot give such guarantees, even though the medians still diverged by a reasonably large margin. These observed trends were further supported by the previously presented Figure 6.6c. While at 500 flows the number of scenarios for which feasible solutions have been found within the time limit was too small for statistical analysis, we still see that using random hints no solutions have been found at all. Using desynchronization hints instead, solutions have been found to up to five of the ten scenarios. We omit a visualization of the results for the small imbalanced topology, as we only recorded differences between executions without hints and with hints as we did for any other topology, but no differences between the various hints themselves.



(a) Median time to first solution

(b) 95% confidence intervals for 400 flows

Figure 6.9: Median time to first solution for the small balanced tree

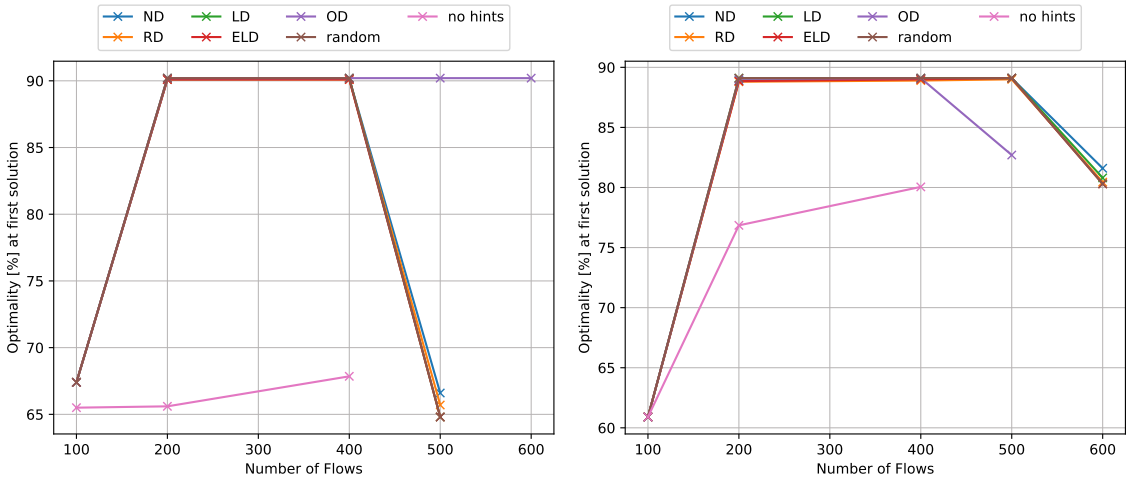
6.5.2 Solution Quality

As we have outlined in the introduction to this thesis, a big advantage of an ILP solver such as Gurobi is the capability of deriving an optimal solution to a given problem. Furthermore, for each solution computed, upper bounds on the optimality of this solution are provided. We will call this upper bound the *optimality gap*. An optimality gap of 0% denotes an optimal solution, while an optimality gap of 100% indicates a comparably poor feasible solution, which diverges from the optimum by a large margin.

Given the optimality gap, we can thus examine the quality of produced solutions by the Gurobi solver. In particular, we want to determine if the provision of hints leads to better overall solutions. To this end, this subsection focuses on the optimality gap at two distinct points during the runtime of Gurobi. On one hand, we are interested in the optimality gap of the first feasible solution found. On the other hand, we are also interested in the evolution of this optimality gap with increasing runtime. Thus, we further examine the optimality gap of the last feasible solution found before the time limit has been reached.

Figure 6.10 shows the median optimality gap for the first calculated feasible solution for the large balanced tree topology and the large imbalanced tree topology. In the previous subsection we were able to augment our data if no feasible solution has been found within the time limit, as we were guaranteed that the actual TTFS was larger than this limit. When inspecting the optimality gap, we cannot make any assumptions for it if no feasible solution has been found within the time limit. Consequently, the presented data only contains scenarios for which at least one feasible solution has been computed. Most noticeably in the two plots is the median optimality gap of the first feasible solution when not supplying any hints. It was significantly better than the optimality gap of solutions produced with provided hints by any desynchronization algorithm.

We observe a trade-off between TTFS and optimality of this first solution. While the TTFS without provision of any hints was significantly higher, this left more time for optimization. When a feasible solution is eventually found, it will be of better quality the longer the TTFS has been. This



(a) Large balanced tree

(b) Large imbalanced tree

Figure 6.10: Optimality of the first feasible solution

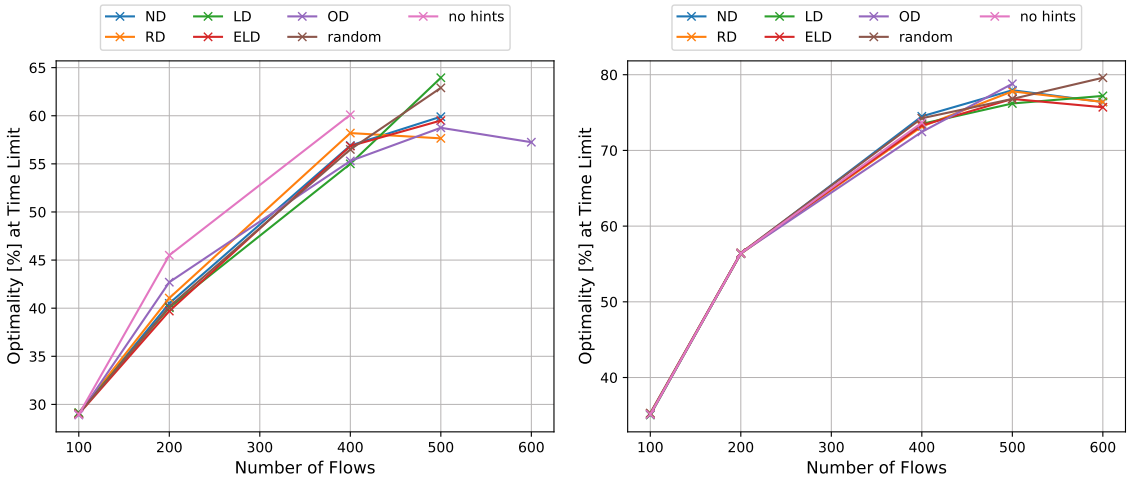
can be further validated by examining the optimality gap for first feasible solutions produced by providing Ordered Desynchronization hints. While on balanced trees the corresponding TTFS was significantly lower than for all other desynchronization approaches, we observed the exact opposite on the large imbalanced tree topology. As a result, on the balanced tree the optimality gap for Ordered Desynchronization is the worst for large test sets, while on the imbalanced tree it decreased noticeably for the test set with 500 flows compared to every other desynchronization algorithm.

We omit a visual representation of the median optimality gap of first feasible solutions for the small topologies, as the obtained results only further support the observations discussed before.

Obviously, the comparison of optimality gaps for first feasible solutions is inherently unfair as we compare two solutions found at different times. Ideally, we want to compare the optimality gap of solutions at a fixed point in time instead.

For this, we inspect the optimality gap of the last feasible solution found when reaching the time limit of two hours (cf. Figure 6.11). Obviously, the solver only accepts new solutions that are better than all previously found solutions. As a result, the last feasible solution found also is the best feasible solution found. We clearly see that after a runtime of two hours the optimality gaps have converged to a similar value for all desynchronization approaches and even for executions without provision of any hints.

We further visualize the evolution of the optimality gap for a single exemplary scenario with 400 flows on the large balanced tree topology (cf. Figure 6.12). Depicted is the optimality gap for an execution without any hints and for an execution with hints. In this example, the hints have been generated by the Relative Desynchronization algorithm. Here we also see the difference in TTFS and the difference in optimality of the first feasible solution. Compared to the execution without hints we observe that the hinted execution generally found solutions of the same quality at an earlier point in time. The closer we get to the set time limit the smaller the difference in solution quality became, until they eventually converge to a similar value, as seen in Figure 6.11.



(a) Large balanced tree

(b) Large imbalanced tree

Figure 6.11: Optimality at the time limit

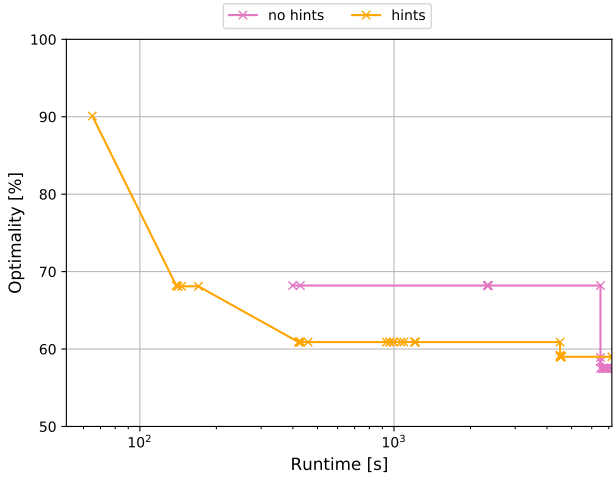


Figure 6.12: Evolution of the optimality gap with increasing runtime

Consequently, we follow that the speed up introduced with the provision of desynchronization hints is most noticeable for early solutions. If the focus lies on obtaining an optimal solution to the No-wait Packet Scheduling Problem, then the provision of desynchronization hints seems optional as our results suggest that the solution quality will eventually converge to similar values over time. We attribute this to the nature of desynchronization. With the presented desynchronization algorithms, we try to approximate a feasible solution to the corresponding NW-PSP. However, as sending times of flows are distributed over the whole cycle, the makespan of this approximated solution will be large. As a result, the desynchronization results approximate solutions of the NW-PSP with a large optimality gap that diverge significantly from the optimal solution. Therefore, especially the calculation of these feasible solutions with comparably low quality is accelerated.

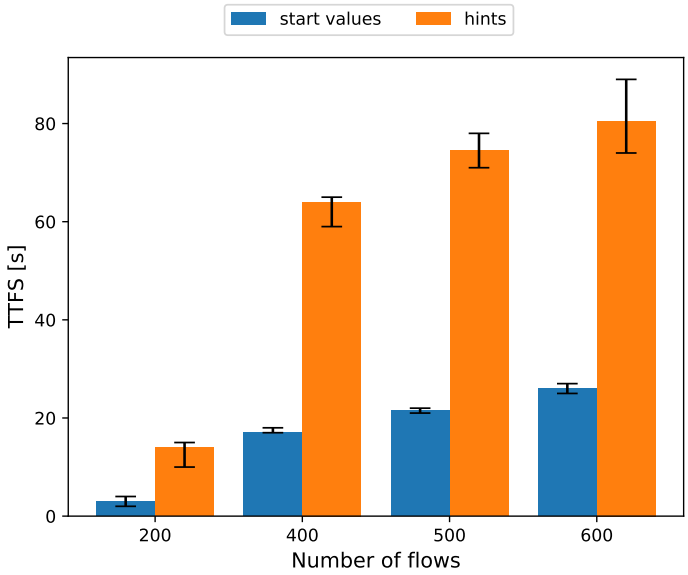


Figure 6.13: Time to first solution - hints and start values

6.5.3 Gurobi Start Values

As an alternative to hints, Gurobi further provides the user with the option to supply so called *start values*. Like hints, they assign initial values to each variable of the ILP. As opposed to hints, the provided start values are required to already form a feasible solution to the problem. Consequently, we see how start values are generally not well suited for the results generated by desynchronization algorithms. However, as we have shown throughout this thesis that Ordered Desynchronization can indeed produce feasible solutions to the corresponding NW-PSP for balanced tree topologies, we were further able to compare the TTFS and optimization process of Gurobi when providing Ordered Desynchronization results as hints or as start values. This comparison is exclusive to balanced tree topologies, as on other topology types Ordered Desynchronization did not compute a feasible solution.

Figure 6.13 shows the median TTFS of the Gurobi solver on the large balanced tree topology. We previously already observed how the median solving time for Ordered Desynchronization hints came with very low variance as the problem was reduced to verifying the hints as a feasible solution. When supplying the desynchronization results as start values, we see how not only the median TTFS was further reduced by a factor of four, but also how the variance decreased even further. As we have no insight into the internal process of Gurobi, we assume that the remaining TTFS can be attributed to the initialization of the ILP variables themselves and is thus independent of the actual scenario. Larger test sets introduce more variables of the integer linear program, which in turn slightly increases the initialization time. Utilizing start values instead of hints however had no impact on the optimality gap. This held true for both the first feasible solution found and the last solution computed before reaching the time limit.

While we saw in the last section that desynchronization hints do heavily improve the TTFS, the results suggested that they do not noticeably speed up the calculation of an optimal solution. The results gathered in this section further indicate that the same holds true for the usage of start

values. If we instead want to calculate some arbitrary feasible solution as fast as possible, we showed that Ordered Desynchronization can act as a standalone scheduling algorithm for balanced tree topologies. As the ILP solver accepted the desynchronization results generated by Ordered Desynchronization as start values, this serves as definite prove for their feasibility. While this renders the usage of an ILP solver in the second phase of the solving process optional, supplying the desynchronization results of Ordered Desynchronization as hints or start values additionally enables us to quickly obtain provable optimality bounds for the respective feasible solution.

6.5.4 Summary

We would like to conclude this evaluation by briefly summarizing the main findings of this section. The primary aspects under examination were the time to the first solution (TTFS) as well as the quality of the computed solutions.

First and foremost, we have observed that the provision of hints generated by our desynchronization algorithms led to a significant speed up regarding the TTFS compared to a Gurobi execution without any hints. For reasonably large test sets we can guarantee a reduction of the median TTFS by at least a factor of six. We further observed that Ordered Desynchronization hints reduced the TTFS for balanced tree topologies the most, as the desynchronization result itself forms a feasible solution to the NW-PSP. For hints that did not form a feasible solution themselves, the presented results for some topologies even suggested not just the provision of hints, but also the quality of the desynchronization influences the TTFS.

On imbalanced trees, we have seen that hints generated by Ordered Desynchronization led to a longer median TTFS compared to the remaining desynchronization algorithms. We could correlate this with the results gathered in the standalone evaluation of desynchronization algorithms. Here we have seen that Ordered Desynchronization both achieved the lowest scores with respect to the introduced desynchronization scores and produced the highest number of collisions. Additionally, at least on the small balanced tree topology we noticed that randomly generated hints led to a smaller speed up than hints generated by the desynchronization algorithms. For the remaining topologies and desynchronization algorithms it was hard to correlate the gathered results with results obtained in Section 6.4, as differences between them could not be considered significant.

We further detected a trade-off between the speed up of the TTFS and the quality of the first computed feasible solution, where a shorter TTFS led to a worse quality of the initial solution. As here we were comparing the quality of solutions found at different points in time, we further examined the evolution of the optimality gap with increasing runtime. The results suggested that hinted executions generally find solutions of the same quality faster than execution without hints. However, this speed up was most noticeable for early feasible solutions with low quality. After reaching the set time limit, the optimality of the current solution has converged against a similar value for all approaches, even if no hints were provided at all. We attributed this to the fact that desynchronization only approximates low quality solutions to the scheduling problem by nature, and thus mainly speeds up the calculation of such solutions.

All in all, we have seen that desynchronization hints are of a great benefit towards the constrained problem, where the only goal is to find some arbitrary solution fulfilling all constraints of the NW-PSP. The TTFS is reduced by at least a factor of six, allowing for a fast computation of a feasible solution with provable optimality bounds.

7 Conclusion

The scheduling problem for Time-Sensitive Networks is proven to be $\mathcal{NP}$-hard. The most prevalent approach to solve such optimization problems remains the formulation of the problem as an *integer linear program*. While there are many similar ILP formulations that model the scheduling problem in Time-Sensitive Networks, we based this thesis on the No-wait Packet Scheduling Problem (NW-PSP). Different heuristics can compute solutions that satisfy all constraints of the NW-PSP. However, an *ILP* solver can solve the given problem to proven optimality, at the expense of very high runtimes. Additionally, it provides provable optimality bounds for each calculated solution.

In this thesis, we adapted the primitive of desynchronization, originally designed for single-hop wireless sensor networks, to the domain of Time-Sensitive Networks. By using desynchronization as a preprocessing step and then relaying the results as input to the ILP solver, we aim at achieving a noticeable decrease in runtime, especially for the computation of a first feasible solution to the problem. By providing the desynchronization results as hints to the Gurobi solver, it is further not required for the desynchronization results to already form a feasible solution to the scheduling problem, but rather approximate how a feasible solution may look like. When adapting desynchronization, we especially saw how certain limitations and concerns to desynchronization for wireless networks, such as the hidden terminal problem, do not apply for Time-Sensitive Networks. However, we also discussed how in turn the different network delays complicate the desynchronization process.

We presented five different desynchronization algorithms for IEEE Time-Sensitive Networks. Two of these approaches aimed to desynchronize from the perspective of the network flows. Here, a flow tries to move its phase as far away from any other flow it shares a common network link with. While Naive Desynchronization only makes a distinction between conflicting and non-conflicting flows and desynchronizes their starting times, Relative Desynchronization additionally takes the network delays into account, by calculating the exact time some flow reaches any conflict link. With this knowledge, instead of desynchronizing the starting times of all flows, Relative Desynchronization tries to maximize the interval between flow transmissions at the point of conflict in the network directly. Two other approaches, Link Desynchronization and Extended Link Desynchronization, tackle the problem from a global perspective of the network, where each link is desynchronized individually, thus, maximizing the interval between any two transmission on the same link. In contrast to both Naive and Relative Desynchronization, these approaches do not try to move a flow as far away as possible from all its conflicting flows, but rather only consider the flows traversing the link currently being desynchronized, which might also be a smaller subset and thus might allow for larger gaps between two flow transmissions. Lastly, our fifth approach, Ordered Desynchronization, tries to exploit differences in the path length of flows up to their conflict link to obtain a specific sending order for which the number of collisions is minimized. For this approach, we have further shown that it will produce a feasible solution to the No-wait Packet Scheduling Problem for balanced tree topologies up to a certain threshold of flows that are to be scheduled.

By both examining the number of collisions and defining two desynchronization scores to measure the quality of the produced desynchronized solutions we were able to first evaluate each desynchronization approach as a standalone algorithm. Here, we have seen that the topology type of the respective network has a large impact on the performance of the desynchronization algorithms. While the two Link Desynchronization approaches and Relative Desynchronization were the least affected by the used topology, there is no desynchronization approach producing perfect solutions across all topologies. We have further validated that Ordered Desynchronization produces feasible solutions to the NW-PSP for problem sizes below a certain threshold. The algorithm can thus be used as a standalone scheduling algorithm for Time-Sensitive Networks for suitable topologies and problem sizes. For evaluated problem instances with up to 800 scheduled flows the desynchronization algorithms produced fast solutions compared to an ILP solver. However, we especially observed that the runtime of Link Desynchronization approaches and Ordered Desynchronization scales noticeably worse than the runtime of the remaining algorithms. Since that runtime is solely influenced by the number of flows, and not by the scheduling complexity of the problem, a utilization of these desynchronization algorithms for problem instances beyond the evaluated size becomes questionable.

When supplying the desynchronization results as input in form of hints to our ILP solver, we have seen a significant speed up compared to an execution without the provision of any hints. The time to a first feasible solution (TTFS) was reduced by at least a factor of six. Also, for balanced tree topologies we have further seen that hints generated by Ordered Desynchronization enabled Gurobi to find a first feasible solution even faster than with hints generated by any other desynchronization approach, as the provided hints themselves already formed such feasible solution. All remaining desynchronization algorithms did not directly produce feasible solutions for the NW-PSP. Still, their hints may vary in quality as some produced desynchronization solutions possibly better approximate a feasible solution to the scheduling problem. When supplying them as hints to the Gurobi ILP solver, results for some topologies, and generally for larger test sets, indicated that the desynchronization quality of infeasible hints does influence the TTFS as well, however, the differences could not be considered significant on most topologies.

The speed up was most noticeable for early feasible solutions with comparably low quality. After reaching the time limit set for a Gurobi execution, the optimality of the current solution converged against a similar value for all approaches, even if no hints were provided at all. The presented results thus lead to the conclusion that the runtime to compute an optimal solution, or generally solutions with very high quality, will not noticeably differ between executions with and without hints. If the goal is to determine some arbitrary solution to the scheduling problem as fast as possible, possibly even within a certain time limit, we see a clear benefit in the provision of hints generated by desynchronization algorithms.

It remains an open question, if the observed trends get more pronounced for test sets with a higher number of flows, larger number of different scenarios and executions without a set time limit. Unfortunately, such extensive evaluations were not possible within the scope of this thesis given the large number of different scenarios that must be executed, each with a considerably high runtime.

Finally, it is yet to be examined how well desynchronization will function for IEEE Time-Sensitive Networks if we move away from the simplified system model that we have assumed for first explorations in this field, especially as heterogeneous network delays and flow sizes invalidate some assumptions made when designing the desynchronization algorithms presented in this thesis.

Bibliography

[BHCW18] H. Boyes, B. Hallaq, J. Cunningham, T. Watson. "The industrial internet of things (IIoT): An analysis framework". **in**: *Computers in Industry* 101 (2018), **pages** 1–12. ISSN: 0166-3615. DOI: https://doi.org/10.1016/j.compind.2018.04.015. URL: https://www.sciencedirect.com/science/article/pii/S0166361517307285 (**backrefpage** 13).

[Bou11] J.-Y. L. Boudec. *Performance Evaluation of Computer and Communication Systems*. EFPL Press, 2011. ISBN: 1439849927 (**backrefpages** 66, 84).

[Cpl09] I. I. Cplex. "V12. 1: User's Manual for CPLEX". **in**: *International Business Machines Corporation* 46.53 (2009), **page** 157 (**backrefpage** 13).

[DK+14] F. Dürr, T. Kohler **andothers**. "Comparing the forwarding latency of openflow hardware and software switches". **in**: (**july** 2014) (**backrefpages** 18, 66).

[DN08] J. Degesys, R. Nagpal. "Towards Desynchronization of Multi-hop Topologies". **in**: *2008 Second IEEE International Conference on Self-Adaptive and Self-Organizing Systems*. 2008, **pages** 129–138. DOI: 10.1109/SASO.2008.70 (**backrefpages** 14, 25, 26, 31, 35, 38).

[DN16] F. Dürr, N. G. Nayak. "No-wait Packet Scheduling for IEEE Time-sensitive Networks (TSN)". **in**: *Proceedings of the 24th International Conference on Real-Time Networks and Systems, RTNS 2016, Brest, France, October 19-21, 2016*. **byeditor**A. Plantec, F. Singhoff, S. Faucou, L. M. Pinho. ACM, 2016, **pages** 203–212. DOI: 10.1145/2997465.2997494. URL: https://doi.org/10.1145/2997465.2997494 (**backrefpages** 13, 14, 17, 19–21, 30, 31).

[DRN07] J. Degesys, I. Rose, R. Nagpal. "DESYNC: Self-organizing desynchronization and TDMA on wireless sensor networks". **in**: **may** 2007, **pages** 11–20. ISBN: 978-1-59593-638-7. DOI: 10.1109/IPSN.2007.4379660 (**backrefpages** 14, 17, 22, 23, 25, 26, 31, 36, 38, 52).

[Gur21] L. Gurobi Optimization. *Gurobi Optimizer Reference Manual*. 2021. URL: http://www.gurobi.com (**backrefpages** 13, 14).

[HGF+20] D. Hellmanns, A. Glavackij, J. Falk, R. Hummen, S. Kehrer, F. Dürr. "Scaling TSN Scheduling for Factory Automation Networks". **in**: *2020 16th IEEE International Conference on Factory Communication Systems (WFCS)*. IEEE. 2020, **pages** 1–8 (**backrefpages** 14, 31).

[Ins16a] Institute of Electrical and Electronics Engineers Inc. "IEEE Standard for Ethernet". **in**: *IEEE Std 802.3-2015 (Revision of IEEE Std 802.3-2012)* (**march** 2016), **pages** 1–4017. DOI: 10.1109/IEEESTD.2016.7428776 (**backrefpages** 13, 17, 18).

[Ins16b] Institute of Electrical and Electronics Engineers Inc. “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— as amended by IEEE Std 802.1Qca-2015, IEEE Std 802.1Qcd-2015, and IEEE Std 802.1Q—/Cor 1-2015)* (**march** 2016), **pages** 1–57. DOI: 10.1109/IEEESTD.2016.7572858 (**backrefpages** 13, 17, 29).

[Ins20] Institute of Electrical and Electronics Engineers Inc. *Time-Sensitive Networking Task Group*. http://www.ieee802.org/1/pages/tsn.html. 1 **july** 2020. URL: http://www.ieee802.org/1/pages/tsn.html (**backrefpages** 13, 17).

[JT11] T. Jensen, B. Toft. *Graph Coloring Problems*. Wiley Series in Discrete Mathematics and Optimization. Wiley, 2011. ISBN: 9781118030745. URL: https://books.google.de/books?id=leL0Y5N0bFoC (**backrefpage** 25).

[KW08] H. Kang, J. Wong. “M-desync: A multi-hop desynchronization for wireless sensor networks”. **in**: *Stony Brook University, Embedded Systems Optimization Lab, Tech. Rep* (2008) (**backrefpage** 27).

[KW09] H. Kang, J. L. Wong. “A Localized Multi-Hop Desynchronization Algorithm for Wireless Sensor Networks”. **in**: *IEEE INFOCOM 2009*. 2009, **pages** 2906–2910. DOI: 10.1109/INFCOM.2009.5062256 (**backrefpages** 14, 25–27, 31, 35, 52).

[MK09] C. Mühlberger, R. Kolla. “Extended Desynchronization for Multi-Hop Topologies”. **in**: (**january** 2009) (**backrefpages** 14, 31).

[MP02] A. Mascis, D. Pacciarelli. “Job-shop scheduling with blocking and no-wait constraints”. **in**: *European Journal of Operational Research* 143 (**february** 2002), **pages** 498–517. DOI: 10.1016/S0377-2217(01)00338-1 (**backrefpages** 14, 19).

[MS90] R. Mirollo, S. H. Strogatz. “Synchronization of pulse-coupled biological oscillators”. **in**: *Siam Journal on Applied Mathematics* 50 (1990), **pages** 1645–1662 (**backrefpages** 22, 31).

[PSRH15] F. Pozo, W. Steiner, G. Rodriguez-Navas, H. Hansson. “A decomposition approach for SMT-based schedule synthesis for time-triggered networks”. **in**: *2015 IEEE 20th conference on emerging technologies & factory automation (ETFA)*. IEEE. 2015, **pages** 1–8 (**backrefpages** 14, 31).

[SL86] C. Sriskandarajah, P. Ladet. “Some no-wait shops scheduling problems: Complexity aspect”. **in**: *European Journal of Operational Research* 24 (3 1986), **pages** 424–438. ISSN: 0377-2217 (**backrefpages** 14, 19).

[ZLZ15] K. Zhou, T. Liu, L. Zhou. “Industry 4.0: Towards future industrial opportunities and challenges”. **in**: *2015 12th International conference on fuzzy systems and knowledge discovery (FSKD)*. IEEE. 2015, **pages** 2147–2152 (**backrefpage** 13).

All links were last followed on May 17, 2021.

Declaration

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

Waiblingen, 17.05.21, P. Schneefuss

place, date, signature