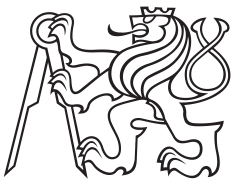
Master's Thesis



Czech
Technical
University
in Prague

F3

Faculty of Electrical Engineering
Department of Control Engineering

Scheduling of the TTEthernet communication

Bc. Martin Heller

May 2016

Supervisor: Ing. Jan Dvořák

České vysoké učení technické v Praze
Fakulta elektrotechnická

katedra počítačů

ZADÁNÍ DIPLOMOVÉ PRÁCE

Student: **Bc. Martin Heller**

Studijní program: Otevřená informatika
Obor: Softwarové inženýrství

Název tématu: **Problém rozvrhování komunikace na protokolu TTEthernet**

Pokyny pro vypracování:

a) Studium specifikace TTTech TTEthernet
b) Studium rozvrhovacích algoritmů pro TTEthernet
c) Návrh sekvenčního rozvrhovacího algoritmu
d) Implementace rozvrhovacího algoritmu
e) Implementace generátoru vstupních instancí
f) Testování a verifikace navrženého algoritmu

Seznam odborné literatury:

[1] Domitian Tamas-Selicean, Paul Pop and Wilfried Steiner, "Design Optimization of TTEthernet-based Distributed Real-time Systems", Real-Time Systems, 2015
[2] D. Tamas-Selicean, P. Pop, W. Steiner, "Synthesis of communication schedules for TTEthernet-based mixed-criticality systems", Proceedings of the 10th International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS 2012), ACM, 2012
[3] Silviu S. Craciunas, Ramon Serna Oliver, and Valentin Ecker, "Optimal Static Scheduling of Real-time Tasks on Distributed Time-triggered Networked Systems", Proceedings of the 19th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA 2014), IEEE, 2014.
[4] Domitian Tamas-Selicean, Paul Pop and Wilfried Steiner, "Timing Analysis of Rate Constrained Traffic for the TTEthernet Communication Protocol," in Real-Time Distributed Computing (ISORC), 2015

Vedoucí: Ing. Jan Dvořák

Platnost zadání: do konce letního semestru 2016/2017

prof. Dr. Michal Pěchouček, MSC.
vedoucí katedry

prof. Ing. Pavel Ripka, CSc.
děkan

V Praze dne 11. 2. 2016

Acknowledgement

I would like to thank to my thesis supervisor Ing. Jan Dvořák for his helpful advice, patience and responsiveness. I would also like to thank to Dr. Alexander Schnell for providing me with his source codes which have been very helpful to me. And last but not least, I would like to thank to my family for their support and encouragement.

Declaration

Prohlašuji, že jsem předloženou práci vypracoval samostatně a že jsem uvedl veškeré použité informační zdroje v souladu s Metodickým pokynem o dodržování etických principů při přípravě vysokoškolských závěrečných prací.

Abstrakt / Abstract

TTEthernet je rozšířením Ethernetu o prostředky pro deterministickou komunikaci. V této práci TTEthernet stručně představíme a uvedeme stávající metody rozvrhování provozu v něm. Následně formulujeme tento rozvrhovací problém jako MRCPSP-GPR (také znám jako multimodální RCPSP/max) a zhodnotíme možnosti použití existujících řešičů MRCPSP-GPR pro rozvrhování provozu v síti TTEthernet. S využitím heuristiky, kterou jsme navrhli, se tento postup jeví jako realistický. Mimo to ještě uvádíme opravu nedávno publikované metody pro odhad maximálního zpoždění rate-constrained (RC) provozu v síti TTEthernet.

TTEthernet is an extension of Ethernet for deterministic communication. We present an overview of TTEthernet and existing methods for scheduling TTEthernet traffic. Then we present a formulation of the scheduling problem as a MRCPSP-GPR (also known as multi-mode RCPSP/max) and evaluate the possibility of using existing MRCPSP-GPR solvers for scheduling TTEthernet traffic. With a heuristic we introduce, this approach appears practical. Apart from this, we present a correction of a state-of-the-art method for estimating worst-case delays of rate-constrained (RC) TTEthernet traffic.

Contents /

Tables /

Chapter 1
Introduction

Time-Triggered Ethernet (or TTEthernet) is an extension of Ethernet for deterministic real-time communication. In this work, we will first introduce TTEthernet in more detail and show what the goals of optimizing TTEthernet traffic can be. Then we present existing approaches to scheduling TTEthernet frames and our own approach. Finally, we present the results of computational experiments performed on our generated benchmark instances and discuss the practicality of our approach for real-world use.

1.1 Motivation

Ethernet (IEEE 802.3) has been the prevalent technology for home and office networks in the last decades. Thanks to its widespread adoption it has developed into a mature technology offering high bandwidth with cheap and readily available hardware.

While unsuitable for critical industrial use in its basic form, the advantages mentioned above drive the efforts to adapt Ethernet for industrial use to supplement and replace some of the currently used technologies. This has become even more pronounced with the ever increasing amounts of data transfers necessary to facilitate features like real-time image processing and recognition or communication among individual units in a smart system. Therefore, extensions of Ethernet are being developed to meet the demands of industrial applications.

TTEthernet is a promising extension of Ethernet, which provides determinism and fault-tolerance while being compatible with standard Ethernet. There has been an ongoing work on the standardization of extensions to Ethernet for scheduled traffic by the IEEE 802.1 Time-Sensitive Networking Task Group and TTEthernet might become a part of the newly developed standard. If substantial changes are made to TTEthernet in the process, the results regarding scheduling traffic in complex multi-hop networks should still be easily applicable.

Determinism with the strictest guarantees is achieved through a fixed schedule for the traffic. Therefore, synthesizing a good (what exactly this means will be discussed later) schedule which meets all the requirements and deadlines is essential for the performance of the network. Because TTEthernet allows for complex topologies, the scheduling involves additional complexity compared to the bus or passive star topologies of older networks like FlexRay.

1.2 Background

Industrial applications like manufacturing process control, automotive, avionics and other critical applications have traditionally carried a very different set of requirements than conventional computer networks.

Conventional computer networks are mainly used for on-demand data transfer without immediate physical impact on the real world. In case of a failure, the transmission

can usually be repeated without causing major difficulties. Therefore, the focus is on bandwidth and efficiency with only moderate demands on reliability.

In contrast, in industrial applications various control loops of physical devices are realized by the network, e.g. data from sensors are transferred to a processing unit which then sends commands to actuators. Any disturbance can thus have immediate effects on the real world with possibly severe consequences. As jitter (variance in transmission times) is detrimental to the function of the control loops, determinism is often required. In addition, individual devices in the network are often limited in hardware and need to operate in demanding environments.

Due to these differences, different technologies and protocols were traditionally used for conventional computer networks and for industrial networks. Ethernet has been used for home and office networks while various Fieldbus networks have been used for industrial applications. However with the increasing integration of the industrial systems, increasing demands on the volume of data transferred and also the maturing and development of the Ethernet, there has been a trend of using Ethernet-based networks for industrial applications as well. An overview of the development of industrial networks is given by [1].

1.3 Ethernet in industrial applications

Ethernet has advanced greatly since its initial standardization in 1983[1]. Data transmission rates have increased from 10 Mbit/s to 100 Mbit/s or 1 Gbit/s with common hardware and to even higher rates with specialized hardware. The introduction of fully duplex links and the transition to switched networks have eliminated collisions on the physical layer and the resulting unpredictable delays due to the exponential back-off algorithm for collision resolution. These developments made Ethernet more suitable for industrial use and several Ethernet-based industrial protocols have been developed. They are based on the Ethernet physical layer and differ in whether they modify the link and network layers, or if they just operate on top of them.

For example, EtherNet/IP (IP stands for Industrial Protocol here, do not confuse with TCP/IP) is an implementation of the Common Industrial Protocol application layer built on top of standard Ethernet and TCP/IP. Thus, it is not (cannot be) strictly deterministic, although real-time performance is achieved through the IEEE 1588 clock synchronization and message prioritization. In contrast, PROFINET IO, which is an adaptation of PROFIBUS onto Ethernet, offers hard real-time capabilities. It uses modified EtherTypes and requires specialized network devices, both as switches and as end systems.

Another application of Ethernet is the AFDX (Avionics Full-Duplex Switched Ethernet) which was designed as a successor to the ARINC 429 single-source multi-drop bus. It allows reducing the amount of wiring reducing the weight of the network infrastructure which is especially important in avionics. Its development was initiated by Airbus for its A380 plane, and it has been used in several other planes including the Airbus A350 or Boeing 787 Dreamliner. From the implementation point of view, AFDX adds traffic shaping and policing to Ethernet to guarantee maximum latencies of the traffic while allowing commercial off-the-shelf Ethernet devices to be used. It does not, however, provide means for scheduled traffic with hard real-time requirements.

[1] http://standards.ieee.org/news/2013/802.3_30anniv.html

The protocols mentioned above have one significant drawback. They are either not strictly deterministic (EtherNet/IP, AFDX) or they are incompatible with standard Ethernet devices (PROFINET IO). Therefore, a new protocol, TTEthernet, which is the subject of this work, has been developed.

Chapter 2
TTEthernet

TTEthernet (TT stands for Time-Triggered) is an extension of Ethernet for deterministic communication. Its development started at the Vienna University of Technology, and the first results were published as TT Ethernet in 2005. In 2008, TTEthernet was introduced by the TTTech company. In 2011 it was standardized as SAE AS 6802 [2].

2.1 Overview

TTEthernet operates at the Level 2 of the ISO/OSI model, above the Ethernet physical layer. It requires a switched network with fully duplex physical links so that unpredictable conflicts when accessing a shared medium are avoided. This also means that a wireless network is not a suitable physical medium for TTEthernet. From a scheduling point of view, however, a fully duplex physical link is equivalent to two separate unidirectional links going in opposite directions. Because it is more convenient to work with unidirectional links, we make this transformation, and from now on, when referring to a *dataflow link* or simply a link we mean one of the directions of the physical link.

Network devices are offered by TTTech. Switches are implemented in hardware, and end systems are implemented in hardware or software (for testing purposes, with limited precision). Unfortunately, we had none of these available for testing and hands-on experience.

TTEthernet specifies a protocol for clock synchronization and the rules for managing the traffic on the network. After an initial startup phase when the clocks of the devices in the network are synchronized for the first time, the operation of TTEthernet is periodic. When in steady operation, the clocks are being periodically synchronized to counter any possible clock drift. This period is called the *integration cycle*.

The messages which follow a deterministic schedule are also periodic. The lowest common multiple of their periods is called the *application cycle* or the *cluster cycle*. The term cluster cycle is used by the AS 6802 standard because the standard allows for the network to consist of multiple clusters, see section 2.5 for more details on the relationship between a cluster and a network. In this work, we assume that the network consists of a single cluster, and we will use the term application cycle consistent with [3] throughout the rest of the work.

2.2 Traffic classes

TTEthernet integrates traffic of different time-criticality levels into one physical network. There are 3 traffic classes in TTEthernet corresponding to the time-criticality levels. These classes, ordered by decreasing priority, are the Time-Triggered (TT), Rate-Constrained (RC) and Best-Effort (BE) traffic. A so-called temporal firewall is employed for separation of the traffic.

2.2.1 Time-Triggered traffic

The Time-Triggered (TT) traffic class has the highest priority, and sub-μs jitter can be achieved (depending on the network devices). TT messages are periodic, in agreement with [4] we assume that they are strictly periodic. Their schedule is calculated offline and then loaded into the individual devices.

The schedule also provides temporal isolation and enables fault tolerance. This is due to the fact that not only the sending of a frame is scheduled, its reception is scheduled as well. If a TT frame arrives outside the time it is supposed to arrive, the *acceptance window*, it is discarded by the receiver. This way even a *babbling idiot* failure, when a faulty device sends spurious frames, can be contained and possible network congestion prevented. This mechanism is called the *temporal firewall*.

2.2.2 Rate-Constrained traffic

For traffic with less strict precision requirements, the Rate-Constrained (RC) traffic class can be used. This traffic class conforms to the ARINC 664p7 specification [5] (also called AFDX). It offers greater flexibility because only the frame routing needs to be determined offline. The messages themselves are event-driven within some limitations.

RC traffic is organized in so-called *virtual links*. As stated in [6], a virtual link is an analogy to the ARINC 429 single-source multi-drop bus. The virtual link determines the routing of the messages associated with it. Furthermore, there are two parameters, L_{max} and BAG, associated with each virtual link. L_{max} is the maximum allowed frame size and BAG is the minimum allowed length of an interval between consecutive frames on the virtual link, called the *bandwidth allocation gap* (BAG). This effectively limits the bandwidth of the virtual link. In exchange for this limitation, the maximum possible delay of any RC message can be calculated offline.

The sending network device needs to ensure that the BAG is kept even if the actual source, e.g. a sensor, sends the messages at irregular intervals. This is called *traffic shaping*. For failure tolerance, the BAG is also enforced on the receiving end, frames not conforming to the BAG are discarded. This is called *traffic policing*.

2.2.3 Best-Effort traffic

Standard Ethernet traffic can also be transmitted through the network. Even standard Ethernet devices, unaware of TTEthernet, can communicate through the network. Such traffic is called Best-Effort (BE) traffic and has the lowest priority. There are no guarantees on the maximum delay of the BE frames and even their delivery is not guaranteed.

2.3 Integration of traffic with mixed time-criticality

When traffic is prioritized using just a simple rule that if there are multiple frames ready to be transmitted, the frame with the highest priority is transmitted first, frames with lower priority can cause delays of frames with higher priority. This happens when a higher-priority frame arrives while a lower-priority frame is in transmission. Then, unless the transmission of the lower-priority frame can be interrupted, the higher-priority frame has to wait until the transmission is finished.

This could cause problems in TTEthernet because random delays of TT frames could be caused by RC frames. Therefore, a method of handling such situations, called the *traffic integration policy*, is needed. The AS6802 standard specifies that a TTEthernet

device shall implement a non-preemptive traffic integration algorithm and may implement other preemptive and non-preemptive traffic integration algorithms.

There are three policies for the integration of TT and RC mentioned by [3] which we characterize below. Of these, only the pre-emption and shuffling are mentioned in the AS6802 standard. However, in our work we will assume the timely block policy is used like in [4] which we base our delay analysis on.

2.3.1 Pre-emption

With the pre-emption integration policy, any transmission of an RC frame which would delay a scheduled TT frame is interrupted, and the TT frame is transmitted exactly as scheduled. The transmission of the RC frame must then be repeated because no RC frames may be lost. The drawback of this policy is that the receiving devices must be able to deal with truncated frames resulting from the possible interruptions. Beware that the truncation cannot be reliably detected at the Ethernet level because, as the AS6802 standard [2] explicitly reminds the reader, "It is possible that the truncation of a frame results in a valid Ethernet frame whose CRC matches its contents."

2.3.2 Timely block

A different approach which causes no extra delay of the TT traffic is the timely block integration policy. In this case, an RC frame can only be transmitted if there is enough time for the transmission of the entire frame before the next TT frame is scheduled. If there is not enough time, the transmission of the RC frame is postponed until after the TT frame is transmitted.

Compared to pre-emption, the timely block policy does not cause truncation of the frames. This comes at the cost of being possibly less efficient because the link is kept idle before the scheduled transmission of a TT frame even if the frame is not actually transmitted (e.g. because it was not sent by the source device).

2.3.3 Shuffling

The above two policies caused no extra delay to the TT traffic, but they also caused inefficient use of the resources. If we allow some delay of the TT traffic, we can wait for the RC frame transmission to finish before we start the transmission of the TT frame. This allows more efficient use of the link at the cost of introducing extra *shuffle_delay* to the TT traffic on each link.

This delay has to be taken into account when scheduling the transmission of a TT frame over the individual links on its path, resulting in an increased end-to-end delay of the frame.

2.4 Network topology

TTethernet allows for a wide range of topologies just like standard Ethernet. The simplest topology would be a set of end systems connected to a single switch in a star topology. Such configuration would be called a simple cluster.

To be able to tolerate a single arbitrary failure not only of any link but also of the switch, another switch would be added to the network and connected to all the end systems, essentially duplicating the interconnection part of the network. This is depicted in fig. 2.1.

If only a failure of a link between the switches needs to be tolerated, the ring topology (see fig. 2.2) can be used. In this configuration, the switch which the sending end system

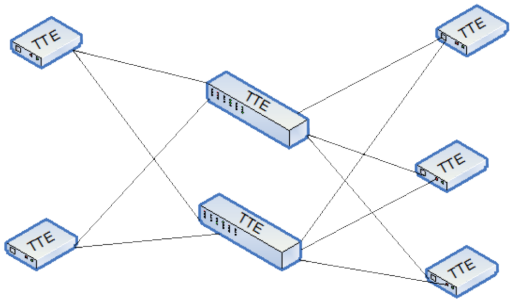


Figure 2.1. TTEthernet dual-channel topology (from [7], p. 10)

is connected to sends the frame in both directions through the ring to the target end system. As the first frame arrives at the switch which the target end system is connected to, the frame is sent to the end system. When the duplicate frame arrives, it is discarded by the switch, so that only one copy of the frame arrives at the target end system.

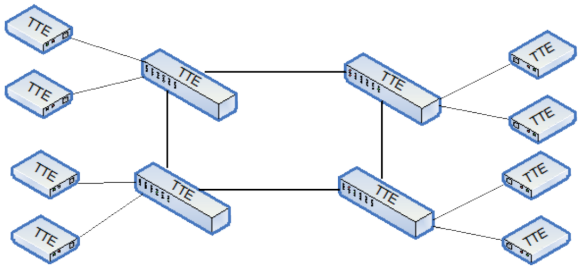


Figure 2.2. TTEthernet ring topology (from [7], p. 10)

TTEthernet can also support various more complex multi-hop topologies, even with varying degrees of redundancy across the network, see fig. 2.3.

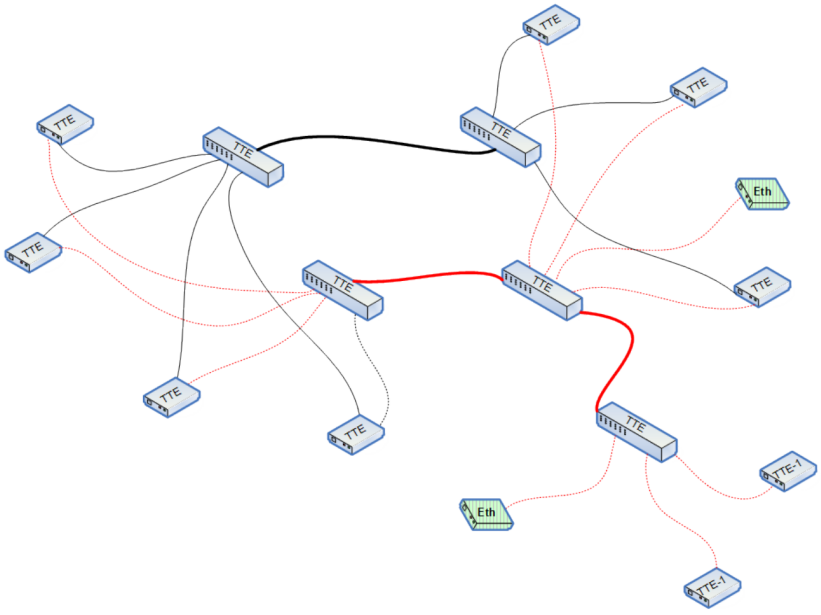


Figure 2.3. Example of a more complex TTEthernet topology (from [7], p. 9)

2.5 Clock synchronization

Clock synchronization, i.e. establishing and maintaining a common global time across the devices in the network is essential for deterministic time-triggered communication. Synchronization is established during a startup phase which we will not discuss because it is not relevant for scheduling the traffic. Once initially synchronized, a clock synchronization service prevents the clocks from drifting too far apart. This synchronization service can be relevant for scheduling because some network capacity needs to be reserved for it. It is also quite interesting so we will discuss it briefly.

2.5.1 Basic concepts

There are several concepts which are important for understanding the synchronization mechanism:

Cluster. Not all devices in the network need to be synchronized to each other; the network can be divided into logical groups called *clusters*. The devices in one cluster are synchronized to each other, and multiple clusters with the same *synchronization domain* may be synchronized to each other based on their *synchronization priority*. Time-triggered communication is only possible among devices within the same synchronization domain. Since we consider networks consisting of a single cluster only, we refer the reader to the AS6802 standard [2] for more details and an explanation of the above terms.

Protocol Control Frames. Ethernet frames of minimum size (64 B) with their EtherType field set to `0x891d` are used for establishing and maintaining synchronization. These are called *Protocol Control Frames* (PCF), and they are sent out at the beginning of each integration cycle. Among other fields, a PCF contains the *pcf_transparent_clock* field where the delay accumulated along its path is stored.

Permanence point in time. To preserve ordering and relative timing of the frames, a transparent clock mechanism, i.e. taking the delay of the PCFs into account, is used. The maximum possible delay of a PCF can be calculated offline, and the fixed delays (e.g. wire delay) are also known beforehand. The actual dynamic delay caused by queuing of the frames as they pass through switches in the network is added to the *pcf_transparent_clock* field of the PCF by each switch. The receiving device can then calculate the actual frame delay and wait for $(max\_delay - actual\_delay)$. At this time instant, it is certain that no other PCF sent before the PCF in question will be received. This instant is called the *permanence point in time* (permanence PIT) associated with the frame.

2.5.2 Synchronization process

The synchronization itself is performed in two phases. In the first phase, a subset of TTEthernet devices, called the *synchronization masters*, sends the PCFs to another set of TTEthernet devices, called the *compression masters*. Each compression master then uses the *compression function* (essentially a median of the received frames permanence PITs) to determine the correct clock value and to correct its clock. PCFs are then sent by the compression masters back to the synchronization masters and to other nodes in the network, called the *synchronization clients*, to let them correct their clocks. This means that the synchronization process will take approximately twice the maximum transmission delay between a synchronization master/client and the compression master plus the compression delay.

Please note that this description is extremely simplified. For a more accurate description explaining how the fault tolerance is achieved see the AS6802 standard.

2.5.3 Comparison to IEEE1588

One might wonder why a new synchronization protocol was introduced when there already is the IEEE1588 standard, and what the advantages of the TTEthernet synchronization mechanism are.

The most significant difference is that unlike in IEEE1588, there can be multiple synchronization and compression masters instead of a single grandmaster clock which all other clocks synchronize to. This eliminates a single point of failure in the network. Even though in IEEE1588 there exists the *Best Master Clock* algorithm for automatic selection of a new grandmaster clock e.g. in the case when the original grandmaster clock is disconnected, this reconfiguration would take several seconds possibly resulting in the clocks drifting too far apart.

In contrast, with multiple synchronization and compression masters and appropriate configuration the TTEthernet network can tolerate the failure of any of the devices and continue uninterrupted operation.

Chapter 3
Scheduling TTEthernet traffic

In this chapter, we are going to discuss what the goals of scheduling TTEthernet traffic are, introduce existing approaches to the scheduling problem, and then we will present our approach to the problem.

Since this is largely an exploratory work, this chapter attempts to capture the process of exploration. Therefore, a precise problem statement is not presented in this chapter; this is left for chapter 5.

We remind the reader that only the TT traffic is scheduled (traffic in the other classes is event-triggered), and that the TT traffic is periodic with the application cycle being the period. Therefore, we will consider scheduling the TT traffic within one application cycle only.

3.1 Goals of scheduling

First of all, the schedule needs to meet all constraints imposed on the TT traffic. These constraints range from basic constraints to more complex application-level constraints imposed by various dependencies among the messages. When considering the constraints we follow [8].

3.1.1 Basic constraints

The basic constraints are needed to ensure the consistency of the schedule. They depend on the network only, and they are independent of the actual application which is running on the network. We will consider the following constraints:

Contention-Free. No two frames can be transmitted on the same link at the same time.

Path-Dependent. A frame can be transmitted over a link in the network only after it was transmitted over the previous link on its path. Actually, a sufficiently large gap is needed between the transmissions to account for the maximum possible clock difference between the devices. If we allowed the shuffling integration policy, the shuffling delay would also need to be taken into account.

Bounded Switch Memory. Switch memory available for TT traffic is bounded, which limits the amount of TT data stored (waiting for the outbound transmission) in the switch. This constraint is simplified by [8] so that the waiting time of any frame in any switch is limited by some constant *membound*.

Simultaneous Relay. According to [8], it can be required by the implementation that when a TT frame is to be transmitted over multiple links from a switch, these transmissions must be performed simultaneously.

3.1.2 Application-level constraints

Apart from the basic constraints listed above, there are constraints given by the actual application. They can involve a single frame, or they can involve multiple frames and introduce dependencies among them.

End-to-end Delay. When e.g. a control loop is realized by the network, it is essential that the delays of the frames are limited for the control loop to function properly. This is expressed by the end-to-end delay constraints, which limit the time between when the frame is sent out by the first device on its path and received by the last device on the path.

Task Interdependence. Because the frames (messages) are generated by tasks which depend on each other, the transmissions of individual frames are also related. This relation is expressed by the task interdependence constraints. For example, a command message for an actuator can only be sent after a message from a sensor was received by the control unit, and after some time needed for processing the message has passed. At the same time, the command for the actuator must be sent no later than $delay_{max}$ after the message from the sensor.

3.1.3 Accomodating RC traffic

When the constraints imposed on TT traffic are met, we can start considering the other types of traffic. What we aim for is minimizing the worst-case delays of RC traffic.

The worst-case delays depend on the amount and parameters of the RC traffic, but also on the TT schedule, which can have a large effect. For example, if the time when the data links are available for RC traffic is too fragmented, the timely block integration policy can cause great inefficiency. To a lesser extent, this would hold for the preemption integration policy too. If on the other hand, we had a very long uninterrupted block of TT traffic, this would also increase the worst-case delays of the RC traffic.

3.2 Existing approach

Since the introduction of TTEthernet, several works on scheduling TTEthernet traffic have appeared. In this section, we will give an overview of these works, which will then lead and inspire us to develop our own approach.

3.2.1 Using an SMT solver

Probably the first publication on the topic of TTEthernet traffic scheduling is [8]. In this work, only the TT traffic is considered, and the goal is to synthesize a schedule which meets the constraints outlined in section 3.1.

Two versions of a scheduler for TT traffic are presented. In the first version, the YICES SMT solver[1] is used out-of-the-box for scheduling up to hundreds of frame instances (a frame instance is the transmission of a frame on a link on its path) in a time limit of 30 minutes.

To allow scheduling of larger problem instances, which the SMT solver could not solve by itself, a second scheduler is presented. This scheduler uses the SMT solver as a backend for scheduling smaller groups of frames, which the frames are divided into. After a group of frames is scheduled, the positions of the corresponding frame instances are fixed, and the next group is scheduled. If inconsistency (the current group cannot be scheduled with respect to the already scheduled frames) is detected, backtracking is used, and the size of the group, which is scheduled at once, is increased.

Using the incremental scheduler, the authors were able to schedule up to 1000 frames (resulting in up to 20000 frame instances), which is close to a practical problem size. It must, however, be noted that the scheduling problem was simplified for the benchmarks.

[1] `http://yices.csl.sri.com/`

Time was divided into slots, each slot corresponding to the transmission of one frame instance, and frames of uniform size and period were assumed.

3.2.2 Schedule porosity

In [9], which builds on and extends [8], traffic from other classes than just the TT class is considered. To reserve capacity for rate-constrained traffic, the concept of *schedule porosity*, i.e. inserting blank slots reserved for the RC traffic into the schedule, is introduced. These blank slots can be introduced *a priori*, *a posteriori*, or by a combination of these two methods.

In the a priori approach, the requirement of having blank slots is added to the constraints the schedule needs to meet. The synthesized schedule will then have these slots reserved for RC traffic. This approach has the disadvantage of introducing additional constraints whose number increases with the number of required blank slots, making the scheduling problem more difficult.

The a posteriori approach attempts to avoid the increase in complexity by creating the schedule first and then inserting the blank slots, essentially by cutting the schedule into pieces and inserting the blank slots between the pieces. However, the end-to-end delay constraints and the task interdependence constraints need to be strengthened before creating the schedule, so that the resulting schedule still meets the original constraints even after the blank slots are inserted.

Both approaches can also be combined in such a way that a small number of larger slots is introduced by the a priori approach, and then a greater number of smaller slots is inserted using the a posteriori approach.

The advantage of the schedule porosity approach is a relatively easy analysis of the worst-case delays of RC traffic thanks to the reserved blocks.

3.2.3 Tabu search heuristic

As noted by [10], porosity scheduling has the disadvantage of the gaps being introduced at the beginning of the scheduling process without considering the profile of RC traffic. Therefore, there is potential for improvement if the RC traffic is considered when creating the schedule.

Such approach is presented in [3] and expanded in [11]. Starting with some feasible schedule, a tabu search algorithm is used to find a schedule such that the worst case end-to-end delay of RC traffic is as low as possible. During the searching process, the schedule is modified using several types of moves like shifting the transmission time of some frame or adding a reserved space for RC traffic. The worst-case delays of RC traffic are calculated using a method adapted from [9].

3.2.4 Combining network-level and task-level scheduling

We have already mentioned when discussing the task interdependence constraints that message processing times play a role in scheduling TT traffic. Several works on this topic have been published recently. In [12], an algorithm for scheduling CPU tasks (the term task is used in a narrower sense here, as a synonym for a processing job), when given a TTEthernet network schedule, is presented. In [13], a method for scheduling both the frame transmissions and the related CPU tasks is introduced. The CPU tasks are actually modeled in a very similar way to the network transmission tasks. A CPU is modeled as another link in the network, a task which needs to run on this CPU is modeled as a frame which needs to be transmitted over this link, and its transmission time is equal to the required CPU processing time.

Because this work focuses on scheduling the network part, we have not explored this direction further. It is, however, a promising area for future work.

3.3 Our approach

Because we had no hands-on experience with TTEthernet, and the AS6802 standard and the secondary sources ([8], [3] and others) gave us only a quite general idea about TTEthernet, we wanted to have our method flexible. This would allow us to adapt the method easily if new information led to a refinement or even a significant modification of our model. Therefore, rather than devising an algorithm for scheduling TTEthernet traffic from scratch, we wanted to make use of existing work on scheduling.

The TTEthernet traffic scheduling problem with all its constraints can be easily formulated as a resource-constrained scheduling problem with generalized precedence relations (RCPSP-GPR, also called RCPSP/max). It is, however, not so clear what objective function should be minimized or maximized in the RCPSP-GPR formulation. For example, minimizing the makespan of the application cycle does not make much sense if there is a periodic frame with a period shorter than the application cycle. It is also not obvious how the RC traffic and its worst-case delay calculation can be integrated into the model.

We will attempt to address the issues mentioned above, transform our scheduling problem to a formulation which can be solved by existing RCPSP solvers, and then we will evaluate the out-of-the-box performance of these solvers. It can be expected that only small instances will be solvable with the most general formulation of the problem. Therefore, we will then attempt to simplify the problem and devise some heuristics aiding the solvers and enabling them to find good schedules for problems of practical size.

3.3.1 Conversion to RCPSP-GPR

Trivially, we can transform the scheduling of one application cycle into an RCPSP instance with generalized precedence relations as follows. Dataflow links become resources with unit capacity (i.e. disjunctive resources). Frame instances become tasks. (Please note that from now on we will be using the term task to describe the elementary activities in the RCPSP.) Each task requires the resource corresponding to the dataflow link the frame instance is transmitted over. The path-dependent constraints are represented by precedences along the frames' paths or by generalized precedences if we want to ensure the gaps between the frame instances.

Using the generalized (i.e. with a minimum time lag specified) precedence relations, we can model all other constraints like the end-to-end transmission deadlines (a negative lag between the tasks corresponding to the last and the first instance of the frame on its path). Similarly, we can model the task interdependence constraints among multiple frames, and the switch memory bound constraints.

If the period of a frame is smaller than the application cycle, i.e. the same frame appears multiple times in the application cycle, we need to add the frame instances for each of the frame occurrences. Since each frame is strictly periodic, there would be generalized precedence relations between the tasks corresponding to the frame forcing them to be exactly *frame_period* apart. The original constraints would need to be duplicated for each set of tasks corresponding to the frame occurrence.

When we want to optimize the schedule for the RC traffic, we can add periodic gaps to the schedule. Depending on the capabilities of the solver we are using, we might just

want to find a feasible schedule or we could attempt more complex optimization. We could, for example, set the gaps to variable length and try to maximize the total length of the gaps. This way, however, the gap sizes could be very irregular resulting in large blocks of TT traffic without any space for RC traffic. Therefore, we might introduce an extra constraint on the minimum length of each gap. Alternatively, if we want the gaps to be as regular as possible, we could set all of them to have identical length and maximize this length. The gaps can be modeled by tasks which occupy all the resources and have time lags set among them which ensure their periodicity.

3.3.2 Alternative: Multi-mode RCPSP-GPR

Although straightforward to construct, the model described above has its drawbacks. Tasks and the relations among them are duplicated for frames with their periods shorter than the application cycle. In addition, there are the precedence relations ensuring periodicity. This creates a complex model which can be difficult to grasp.

For an alternative approach, we draw inspiration from [14], which deals with scheduling time-triggered traffic on the FlexRay bus. Like in TTEthernet there is an application cycle which consists of multiple integration cycles, in FlexRay there is a hyperperiod which consists of multiple communication cycles. FlexRay messages are strictly periodic too, i.e. if a message appears multiple times in the hyperperiod, its offset in the communication cycle must always be the same. It is specific to FlexRay that the communication cycle is divided into a static segment, which contains only scheduled traffic, and a dynamic segment, which contains event-triggered traffic. The goal of scheduling as presented in [14] is to minimize the maximum length of the static segment across the communication cycles. When visualizing the schedule, it is therefore convenient not to display a single timeline for the whole hyperperiod. Instead, multiple aligned timelines are displayed, one for each communication cycle, see fig. 3.1.

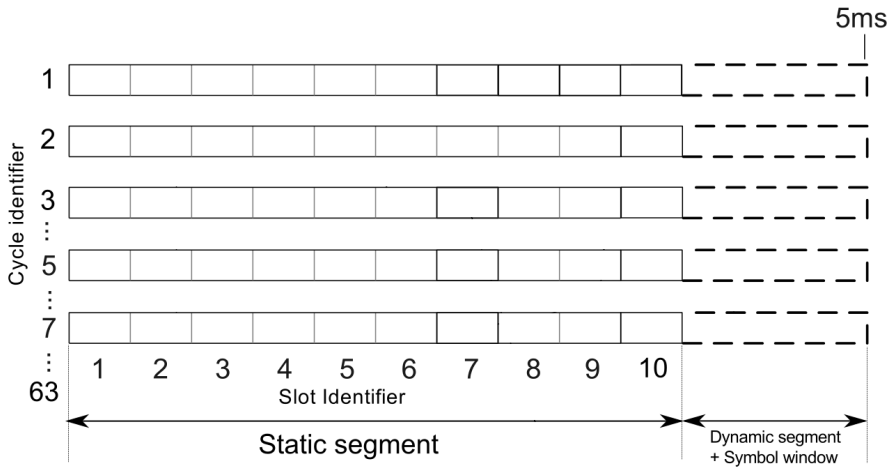


Figure 3.1. FlexRay timeline visualization (adapted from [14], p. 2)

We can interpret each timeline as a renewable resource with unit capacity, which the tasks are using. Now, instead of one resource corresponding to the FlexRay bus during the hyperperiod, we have multiple resources corresponding to the FlexRay bus during communication cycles $0, 1, \ldots, N-1$. Then, instead of having multiple tasks corresponding to multiple occurrences of a message in the hyperperiod, we can have a single task which occupies multiple resources. For example, if we had a hyperperiod

of 6 communication cycles, the task corresponding to a message with a period of 2 communication cycles could occupy resources 0, 2 and 4. However, it could also occupy resources 1, 3 and 5 if we decide to schedule it later in the hyperperiod. Therefore, while reducing the number of tasks which need to be scheduled, additional complexity is added because the tasks can be executed in alternative modes with varying resource demands. This leads to the multi-mode RCPSP formulation. The optimization criterion, minimizing the length of the static segment, then turns into minimizing the makespan of the multi-mode RCPSP.

With TTEthernet, we can apply an analogous transformation for each dataflow link and the frame instances belonging to it. Let $T_{appcycle}$ be the length of the application cycle and T the length of the integration cycle. For each dataflow link, we will have $\frac{T_{appcycle}}{T}$ resources. Like in the single-mode RCPSP formulation, the frame instances become the tasks, but there won't be any repetition of the frame instances. Instead, let f be any TT frame in the network and T_f its period. Then every task $\tau_f^{(j)}$ corresponding to the transmission of frame f over some dataflow link $dl^{(j)}$ has $\frac{T_{appcycle}}{T_f}$ modes, and each mode represents the offset (in integration cycles) of the task within the application cycle. Let $r_0^{(j)}, r_1^{(j)}, \ldots, r_{\frac{T_{appcycle}}{T}-1}^{(j)}$ be the resources corresponding to $dl^{(j)}$. Then the task $\tau_f^{(j)}$ occupies resources $r_m^{(j)}, r_{m+\frac{T_f}{T}}^{(j)}, \ldots, r_{m+k\cdot\frac{T_f}{T}}^{(j)}$ where $k = \frac{T_{appcycle}}{T_f} - 1$, when executed in mode m. Relating this to the previous example, in mode 0 the task occupies resources 0, 2 and 4, and in mode 1 it occupies resources 1, 3 and 5.

The path-dependent and other constraints are then expressed using the generalized precedence relations between the tasks. However, now that we have introduced the modes in which tasks can be executed, the path-dependent constraints have become more complicated. Clearly, if we have two consecutive links on a data path, the frame cannot be transmitted in cycles 1, 3 and 5 over the first link and in cycles 0, 2 and 4 on the second link. Therefore, it must be ensured that *link2_mode* $\geq$ *link1_mode* (using an informal notation to avoid verbosity or introducing an excessive number of symbols). To avoid the need of transforming the generalized precedence relations, we have strengthened the constraint to *link2_mode* = *link1_mode*, i.e. we require that every frame is transmitted end-to-end within one integration period. This can be expressed using the generalized precedence relations by setting very large time lags between incompatible modes. In retrospect, it seems that the transformation of the generalized precedence relations might be actually quite straightforward, and that the restriction is therefore unnecessary. We leave this for future work.

We can see that when using the lags to enforce the relations between the tasks' modes, the lags depend on the modes the tasks are executed in. When *link2_mode* = *link1_mode*, we keep the original lags, otherwise, we replace them with a very large lag between the transmission on *link1* and *link2*. These very large lags will ensure that when combined with some upper bound on the makespan, the unwanted changes of modes along a frame's path will be infeasible. As a result, we get a multi-mode resource-constrained project scheduling problem with generalized precedence relations (MRCPSP-GPR).

Now we have to discuss the objective function we are going to minimize or maximize. As mentioned earlier, we would like to make the TT schedule as favorable for the RC traffic as possible in order to minimize its worst-case end-to-end delay. Unfortunately, this criterion cannot be easily expressed using the available scheduling tools, and it is quite costly to compute.

What MRCPSP-GPR solvers are well-suited to, is finding the minimum makespan (C_{max}) of the schedule. This is analogous to minimizing the length of the static segment in a FlexRay schedule, and it leads to the creation of gaps reserved for RC traffic at the end of each integration cycle. If the traffic is relatively balanced among the links, and if no constraints force some tasks near the end of the integration cycle, this approach works well. If this is not true, the minimum makespan will be determined by some task near the end of the integration period, and we have no control over the rest of the schedule.

Therefore, we can also attempt to insert multiple gaps into the integration cycle, and maximize the sum of their lengths while requiring some minimum size of each gap. We could let these gaps float in the integration cycle, or we can set them to fixed positions. We could also set them to have equal length and maximize this common length. The theoretical possibilities are vast, and they depend on the capabilities of the solvers which we will use as a backend. We will introduce the solvers in chapter 4, and then we will present a formal problem statement for the TTEthernet scheduling problem.

This returns us to the topic of solving the formulated problem. Solving MRCPSP-GPR optimally is very hard in general (NP-hard), with state-of-the-art methods solving some relatively small ($\sim$ 100 tasks) instances only recently, see [15]. For practical use, however, finding a "good enough" solution is usually sufficient without the need to prove optimality. Furthermore, we hope that the solvers will be able to exploit some structure which is present in the formulation so that larger problems can be solved.

Thus, we will evaluate the out-of-the-box performance of the solvers first, and if it is not sufficient to solve problems of practical size, we will attempt to create heuristics and make assumptions which would make the problem more tractable.

3.3.3 Mode assignment heuristic

We have observed that the difficulty of solving the scheduling problem increases greatly when multiple modes of task execution are introduced. Therefore, we would like to select and fix the modes in advance, to convert the problem back to a single-mode RCPSP-GPR.

As the resource requirements of tasks vary by their modes, it is natural to think about balancing the loads of the resources, i.e. minimizing the load of the busiest resource. When transformed back to the TTEthernet scheduling problem, this means minimizing the maximum amount of TT traffic which needs to be transmitted over any link in a single integration cycle.

When performing the load balancing, we ignore all the precedence relations except the large lags forbidding the combination of incompatible modes. For the balancing itself, we use the CPLEX MIP solver from the IBM ILOG CPLEX Optimization Studio.

Chapter 4
MRCPSP-GPR Solvers

To solve the formulated scheduling problem, we need a good solver, and actually, there are not many solvers designed for solving MRCPSP-GPR. Originally we found just one MRCPSP-GPR solver by Schnell and Hartl [16] based on the SCIP constraint integer programming framework [17]. Later we found out that the IBM CPLEX CP Optimizer [18] offers powerful tools for scheduling, which make it easy to express scheduling problems and offer a lot of flexibility.

In this chapter, we will introduce the solvers in more detail and outline the principles they are based on.

4.1 Constraint programming

Both the solvers use constraint programming in their cores. Constraint programming is a technique which expresses the relations between variables using various constraints. Unlike in linear programming, the constraints need not be linear; they can be more complex. On one hand, this makes the solution process much harder, on the other hand, these general constraints can be much more expressive and better in capturing the nature of the problem. It also enables a concise and elegant formulation of the problem.

For example, if we have the `cumulative` constraint which, given the tasks' start times, processing times and renewable resource requirements, ensures that the resource capacity is not exceeded at any moment, the single-mode RCPSP can be modeled as follows (taken from [19]):

$$\begin{aligned} \min \quad & \max_{j \in \mathcal{J}} (\boldsymbol{S}_j + \boldsymbol{p}_j) \\ \text{s.t.} \quad & \boldsymbol{S}_i + \boldsymbol{p}_i < \boldsymbol{S}_j \quad \forall i \text{ preceding } j \\ & \texttt{cumulative}(\boldsymbol{S}, \boldsymbol{p}, \boldsymbol{r}_{\bullet k}, \boldsymbol{R}_k) \quad \forall k \in \mathcal{R} \\ & \boldsymbol{S}_j \geq 0 \quad \forall j \in \mathcal{J} \end{aligned}$$

where $\mathcal{J}$ is the set of tasks, $\mathcal{R}$ is the set of resources, $\boldsymbol{S}$ is the vector of the tasks' start times, $\boldsymbol{p}$ the vector of their processing times, $\boldsymbol{r}$ the matrix of the resource requirements, and $\boldsymbol{R}$ is the vector of resource capacities.

There are two essential concepts in constraint programming, constraint propagation and conflict analysis, which we will explain in more detail.

4.1.1 Constraint propagation

To avoid exhaustively searching the space of solution candidates, as it would be necessary if the constraints only provided a feasible/infeasible answer for a given point, extra information is usually provided by the constraints. This information can then be used to tighten variable bounds or strengthen the constraints. This is called *constraint propagation*.

To illustrate this, let's assume that when solving the RCPSP, we arrive at the following situation. Originally, we knew that $task_1$ had to be scheduled so that it started after some s_{min} and finished before some s_{max}. Then, $task_0$ was scheduled at s_0 such that there is not enough time between the finish time of $task_0$ and s_{max} for the execution of $task_1$. Therefore, the upper bound on the finish time of $task_1$ is lowered to a new $s'_{max} = s_0$. This is illustrated in fig. 4.1.

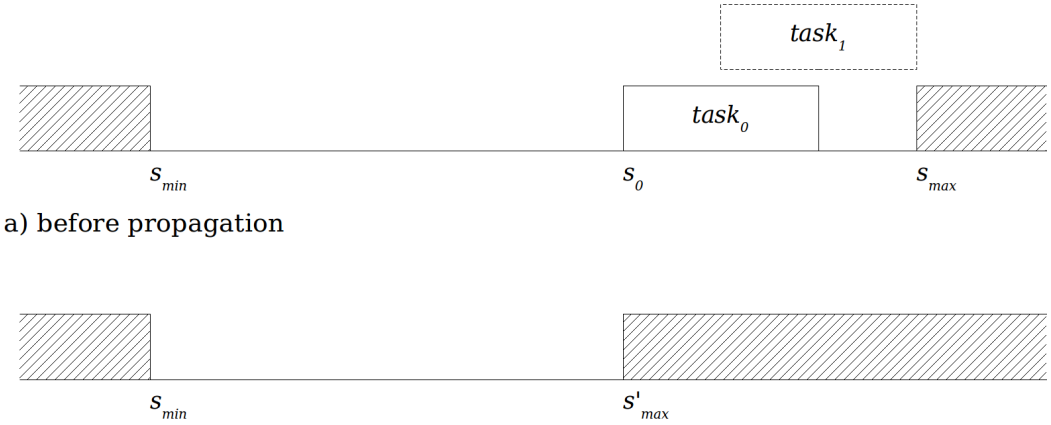


Figure 4.1. Example of constraint propagation

4.1.2 Conflict analysis

Constraint propagation by itself would be sufficient to solve the problem only in very lucky cases. Normally, branching on variables and backtracking is needed to make progress. However even for small problems, the space to be searched is very large and impossible to be explored in whole.

The number of states which need to be searched can be greatly reduced if an infeasibility is detected early, and conflict analysis enables such early detection. Originally used by SAT solvers to detect infeasible subproblems, this technique has been extended to mixed integer programming by Achterberg [20].

When the bounds of a variable change, it is recorded what (which variables' bounds) led to the change. When an infeasibility is detected, an attempt is made to construct a constraint which captures the reason for the infeasibility. This constraint then serves to detect and discard similar situations later in the search.

4.2 SCIP-based MRCPSP-GPR solver

The MRCPSP-GPR solver presented in [16] is built on top of the SCIP constraint integer programming framework [17]. SCIP is a framework which provides tools for implementing branch-and-bound based algorithms, introduced by Achterberg [21]. It also includes many algorithms controlling the searching process in the form of plugins. These plugins enable SCIP to be used as a MIP solver out-of-the-box or to easily create models using prepared constraints.

SCIP is open-source, and it is licensed under the ZIB Academic License [22] which allows using the licensed work by members of non-commercial academic institutions. It also allows modification and distribution under the same license.

4.2.1 Constraint Integer Programming

The SCIP framework is built around the concept of constraint integer programming. It integrates constraint programming (CP), SAT and mixed integer programming (MIP) techniques in an attempt to make use of their strengths and to mitigate their weaknesses. To allow this, a restriction of CP called the constraint integer programming (CIP) is proposed in [23]. CIP restricts the objective function to a linear function and further it requires that after fixing all integer variables, the remaining problem is a linear program. A formal definition is given in [23].

This restriction is not overly restrictive as clearly every CP problem with a finite domain can be expressed as CIP. The linearity restriction on the objective function can be easily circumvented by introducing an auxiliary variable which is bound to the actual objective function by a constraint which can be non-linear. Moreover, clearly, every MIP problem is also a CIP problem.

A diagram showing the operation of SCIP integrating the CP and MIP techniques is shown in fig. 4.2. The program flow is directed by so-called constraint handlers which we will briefly introduce.

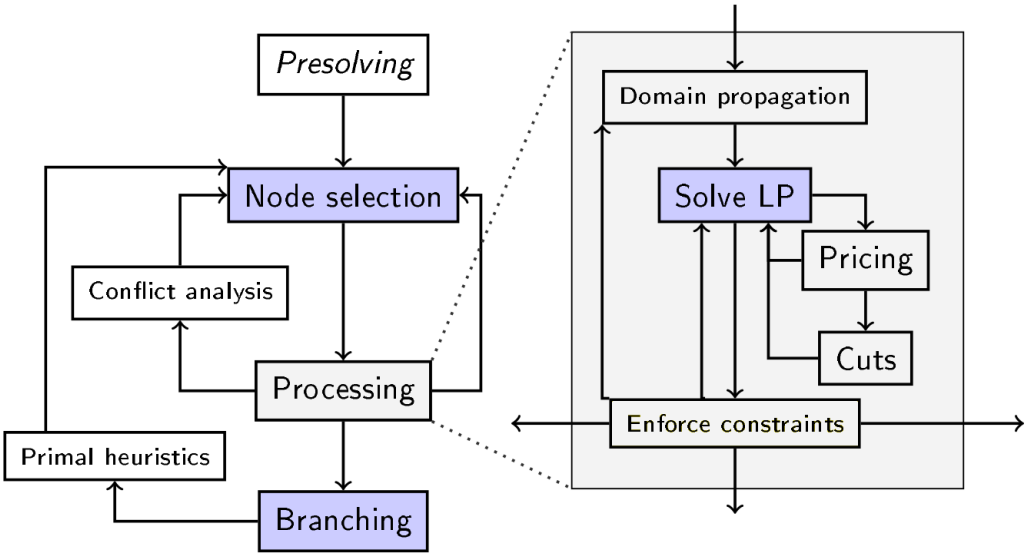


Figure 4.2. Diagram of SCIP operation (from [24], p.21)

4.2.2 SCIP constraint handlers

Actual classes of constraints are represented by SCIP plugins called *constraint handlers*. Many constraint handlers, e.g. for the linear, knapsack or cumulative constraints, are already included in the SCIP Optimization Suite, and these can be easily used for modeling a problem. Custom constraint handlers which may be able to capture the structure of the problem more efficiently can be implemented in C or C++.

A SCIP constraint handler is a set of callback methods (or an object implementing these methods if the C++ API is used), which are called at various phases of the search and guide the search process. There are four callback methods, the CONSCHECK, CONSENFOLP, CONSENFOPS and CONSLOCK, which every constraint handler has to implement.

CONSCHECK just checks the provided solution and returns a simple feasible/infeasible answer. CONSENFOLP or CONSENFOPS are called in the Enforce

constraints (see fig. 4.2) phase. A solution of some relaxation of the problem is provided to the callback to be checked for feasibility. Unlike CONSCHECK, however, not just a simple answer "infeasible" is returned in the case of infeasibility. Instead, these callbacks can attempt to resolve the infeasibility e.g. by adding some constraint, by reducing the domain of some variable or by branching. The last of these callbacks, CONSLOCK, serves to indicate if and in which direction the values of the variables can be changed without violating the constraint.

If a constraint handler provided these callbacks only, the constraint would be enforced but this would not be very efficient as no guidance for the search process would be available. Therefore, other callback methods for domain propagation, conflict resolution, adding extra cutting planes and many other situations can be implemented. Some information on how to create a custom constraint handler is given in the documentation of SCIP [25] and in materials from the Combinatorial Optimization at Work workshop [26] where a solved example is also available. A few examples are also included in the SCIP Optimization Suite.

We must note that although many constraint handlers are available for study, learning how to create a good SCIP constraint handler is far from being easy. There is quite a gap between the fairly simple example constraint handlers with less than 1000 lines of code and the constraint handlers for practical use which have up to more than 10000 lines of code (`cons_linear` having 17321 LoC, `cons_cumulative` 14417 LoC in SCIP 3.2.0).

What is important to be aware of is the use of macros. Constraint handler callbacks are declared and defined using predefined macros, effectively hiding the methods' signatures including parameter names and this can be quite confusing. We found it more readable when another function whose signature was not hidden was called from the callback, and all the useful parameters were passed to it. Similarly, the `SCIP_CALL` macro may look intimidating at the first sight, as if SCIP functions were called in some special way. In fact, it is just a utility macro which checks the return value of the function, and propagates the value by returning it early in case the value indicates an error.

4.2.3 CumulativeMM

In section 4.1 we have already met the `cumulative` constraint which, for single-mode tasks, ensures that at any given moment the demand for the specified resource does not exceed the resource's capacity. SCIP includes the `cons_cumulative` constraint handler in its distribution, which corresponds to the `cumulative` constraint.

Like there was one `cumulative` constraint for each renewable resource in the RCPSP formulation example, one `cons_cumulative` constraint handler instance is created for every resource. The constraint handler operates on integer variables representing the start times of individual tasks. The tasks' durations, their resource demands and the resource's capacity are assumed constant, and they are passed to the constraint handler on construction.

For multi-mode tasks, where the resource demands vary by mode, Schnell and Hartl [27] introduced the `cons_cumulativemm` constraint handler. In addition to the integer variables representing the tasks' start times, this constraint handler operates on binary variables representing the assignment of modes to the tasks. The tasks' durations and resource demands are then given for every task-mode pair.

Apart from the compulsory callbacks, this constraint handler implements a domain propagation callback – CONSPROP, and a conflict analysis callback – CONSRESPROP. These callbacks are based on the concept of *task cores*. A core of a task

is a part of the task which is certain to run during a certain time period and occupy a certain amount of the resource.

For example, let's have a task whose duration in mode 1 is 10 and in this mode it occupies 3 units of the resource. In mode 2 its duration is 8 and it occupies 5 units of the resource. Then, even when we do not know which mode the task will be executed in, we know that the task's duration will be at least 8 and its resource requirement at least 3 units. In addition, we know that the task can start at time 0 at the earliest, and must be finished by time 14. Then we know that regardless of its mode, it needs to run between time 6 and 8, and it will occupy at least 3 units of the resource, and we call this the core of the task. This allows us to create some resource consumption profile for each resource even when the tasks' start times and modes are not yet known. Figure 4.3 illustrates how the task core is constructed in the above example. For simplicity, we only show the varying start time of the task.

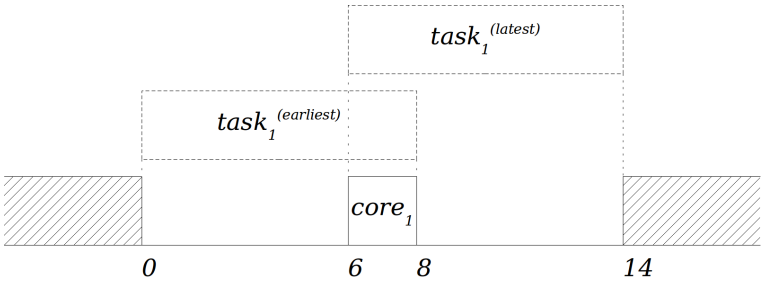


Figure 4.3. Example of constructing a core of a task

Then we can consider individual tasks whether they can, in their minimum duration and resource consumption across their allowed modes, fit into the profile at their current minimum and maximum start times. If not, we can move the bounds on the task's start time and thus reduce the variable's domain, as we have shown in section 4.1.1. In addition, it is recorded which other task was the reason for the change of the bounds. If infeasibility is detected, this information can then be used by SCIP conflict analysis mechanism to generate a constraint to avoid this configuration in the future.

The code of the constraint handler has been kindly provided to us by Dr. Alexander Schnell. We studied the code with the aim of modifying it to improve its performance either in general or at least in some special cases which would appear when solving our scheduling problem. This turned out to be more difficult than originally expected due to the nature of SCIP as described earlier. By fixing an omission in the code and by treating the case of resources with unary capacity (disjunctive resources) separately we have managed to improve the performance slightly. However, the improvement was not proportionate to the effort. Therefore, we have abandoned this effort after finding out that the IBM CP Optimizer was greatly superior for our problem and becoming competitive was unlikely even if some further improvements could be made.

As a side remark, we would like to warn the reader against attempting any optimizations without fully understanding what is happenning. This way, we attempted to optimize the bounds propagation of the tasks' start times. When propagating the bounds as in fig. 4.1, the bound may be shifted by a large value if there are multiple tasks like $task_0$ already scheduled. However, in the code, this shift is always performed by at most the task's processing time, and done iteratively if needed. It seemed to us as a good idea to perform the shift in one large step. At first, this led to a quite significant speedup, but eventually, we found out that the results were incorrect. Only

then have we realized that the iterative propagation in the small steps was essential for the conflict analysis. For each small step, we can say that it was caused by the original value of the bound and by the position of the conflicting task. The aggregated shift of the bound is, however, not caused by any single one of the conflicting tasks. When our faulty optimization failed to take this into account, it led to the incorrect results.

4.2.4 GenPrecMM

The precedence relations and time lags are handled by the `cons_genprecmm` constraint handler. Unlike the `cons_cumulativemm` constraint handler whose instance is created for every renewable resource, only one instance of the `cons_genprecmm` constraint handler is used for the whole model. The constraint handler operates on the task start integer variables and the task-mode assignment binary variables, and takes the table of lags between individual job-mode pairs as a parameter. Like the `cons_cumulativemm` constraint handler, it implements the domain propagation and conflict analysis callbacks.

After the experience with the `cons_cumulativemm` constraint handler, we have not studied `cons_genprecmm` in much detail, and we have been using it essentially as a black-box.

4.3 IBM CPLEX CP Optimizer

The IBM CPLEX CP Optimizer, a part of the IBM ILOG CPLEX Optimization Studio, is a tool for modeling constraint programming problems, especially scheduling. We actually first learned about its scheduling capability from a benchmark [15] where the most recently closed (presumably the most difficult) instances have been closed using a technique called failure-directed search [28], which is integrated into the CP Optimizer.

Because the CP Optimizer provides all the tools we need out-of-the-box, we have not examined the principle of its operation, we have just used it as a black-box. Therefore, we will describe its features from a user's point of view.

Apart from an IDE and the OPL problem specification language, the CP Optimizer offers APIs for .NET framework languages, C++ and Java. We have used the C++ API. For modeling with the C++ API, a so-called environment of type `IloEnv` is needed. Inside this environment, the model of type `IloModel` and all the variables are created.

4.3.1 Scheduling tools

The CP Optimizer provides convenient tools for scheduling, which also enable it to perform scheduling-specific optimizations. Tasks are not represented just by their start times and fixed durations. Instead, they are represented by interval variables, `IloIntervalVar` in the C++ API, which the constraints operate on. The interval variables allow expressing the constraints in a more straightforward way.

For example, the resource demand of a task is represented by an `IloPulse` expression which translates into a function that has a value of zero outside the interval and a specified value inside the interval. Multiple `IloPulse` are then added up into an `IloCumulFunctionExpr` expression. The resource capacity constraint is then represented by a simple arithmetic expression involving the `IloCumulFunctionExpr` expression. For example, let's have n tasks represented by the interval variables stored in an array (type `IloIntervalVarArray`) `taskIntervals` with their resource demands stored in an array `resourceDemands` and durations in an array `durations`. Then the cumulative

constraint due to the capacity of the resource `resourceCapacity` can be expressed as follows:

```
IloEnv env;
IloModel model(env);

IloIntArray resourceDemands(env, n);
IloIntArray durations(env, n);
IloInt resourceCapacity;

/* initialize the values */

IloIntervalVarArray  taskIntervals(env, n);
IloCumulFunctionExpr resourceUsage(env);

for (int i = 0; i < n; i++) {
    taskIntervals[i] = IloIntervalVar(env);
    taskIntervals[i].setSizeMin(durations[i]);
    taskIntervals[i].setSizeMax(durations[i]);

    resourceUsage += IloPulse(taskIntervals[i], resourceDemands[i]);
}
model.add(resourceUsage <= resourceCapacity);
```

If we have a resource with unary capacity and tasks with unary demands, i.e. a disjunctive resource, we can use the `IloNoOverlap` constraint instead. This constraint, as its name suggests, prevents the intervals from overlapping.

```
IloIntervalVarArray  resourceUsers(env);
/* fill the resourceUsers array with tasks from taskIntervals
   with nonzero resource demand */
model.add(IloNoOverlap(env, resourceUsers));
```

Precedences between intervals can be expressed by various constraints like `IloStartBeforeStart` or `IloEndBeforeStart`. These constraints also allow their user to specify the minimum allowed time lag between the intervals. For example, if we require a minimum lag of length 123 between the start of the `first` and `second` intervals, we can ensure it as follows:

```
IloIntervalVar first(env);
IloIntervalVar second(env);
IloInt minLag = 123;

model.add(IloStartBeforeStart(env, first, second, minLag));
```

To allow easy modelling of different modes a task can be executed in, the interval variables can be set to *optional*. This means that the task may or may not be present in the schedule, and its presence depends on the constraints and the objective function. An array of optional interval variables can then be bound to another interval variable using the `IloAlternative` constraint. It ensures that this interval is present if and only if exactly one of the optional intervals is present. Let's assume that a task has to be executed in exactly one of m modes. This can be modeled as follows:

```
IloIntervalVar taskInterval(env);
IloIntervalVarArray  modeIntervals(env, m);

for (int i = 0; i < m; i++) {
    modeIntervals[i] = IloIntervalVar(env);
    modeIntervals[i].setOptional();
}
model.add(IloAlternative(env, taskInterval, modeIntervals);
```

When dealing with mode-dependent resource consumption and time lags between the tasks, we can now work with the intervals corresponding to the modes without worrying which one is present. For example, the IloPulse translates to zero when the interval it is based on is not present. Similarly, a precedence constraint like `IloStartBeforeStart` is automatically disabled if any of the intervals it operates on is not present.

Apart from the specialized constraints for scheduling introduced above, the CP Optimizer also offers ordinary arithmetic constraints which can be applied to various expressions. For example, the constraint on the minimum lag between the starts of the intervals

```
model.add(IloStartBeforeStart(env, first, second, minLag));
```

could also be expressed as

```
model.add(IloStartOf(first) + minLag <= IloStartOf(second));
```

When possible, it is, however, very important to use the specialized scheduling constraints instead of these arithmetic constraints. We experienced this when implementing the generalized precedence relations. Because we had overlooked the optional argument to `IloStartBeforeStart` specifying the minimum time lag between the intervals, we created arithmetic constraints between the start times of the intervals just like the one above. When combining this with the optionality of intervals, it led the solver to run for a long time (the actual length is unknown because we aborted the computation after several minutes) without producing any output or progress status.

After asking about this issue on the CP Optimizer support forum[1], we learned from Philippe Laborie (IBM) that this is probably due to an extremely slow domain propagation. When we followed his advice and switched to `IloStartBeforeStart` with the lag specified as the extra argument, the problem has disappeared.

[1] https://www.ibm.com/developerworks/community/forums/html/topic?id=973401be-d891-4215-a260-a5706fb17ee3

Chapter 5
Problem summary

In the previous chapters, we have introduced TTEthernet, we have discussed possible approaches to scheduling TTEthernet traffic, and we have presented some tools which can be used for this purpose. We have, however, not yet specified what exact approach we choose, and what our objective function will be. We are going to fill this gap now.

Because we find this formulation the most elegant and easy to work with, we formulate the problem of scheduling TTEthernet traffic as a multi-mode RCPSP with generalized precedence relations (MRCPSP-GPR) as described in section 3.3.2. That is, we won't use the direct RCPSP-GPR formulation where the tasks corresponding to frame instances are scheduled within the application cycle of length $T_{appcycle}$, and the resources correspond to the links in the network. Instead, we will schedule the tasks within one integration cycle of length T only, and the resources will correspond to links in the individual integration cycles, i.e. we will have $\frac{T_{appcycle}}{T}$ resources for each link.

Each task then requires the resources corresponding to the integration cycles the frame instance is transmitted in. When a frame's period T_f is longer than the integration cycle, there are $\frac{T_f}{T}$ possibilities for the set of the integration cycles the frame instance is transmitted in. Therefore, each task corresponding to the frame f can be executed in one of $\frac{T_f}{T}$ modes, each mode requiring a different set of resources. To simplify the formulation, we impose a restriction that all tasks corresponding to one frame have to be executed in the same mode. In terms of the original problem, this means that every TT frame has to be transmitted end-to-end within a single integration cycle. The offset of the frame instance within the application cycle is then determined by the offset of its corresponding task within the integration cycle, and by the mode the task is executed in.

The path-dependent constraints are expressed by minimum time lags between the transmissions on successive links in the data path. The restriction that all tasks corresponding to one frame have to be executed in the same mode is expressed using a very large lag between these tasks when executed in different modes. The other constraints are expressed similarly using the time lags. Deadlines are expressed using negative time lags.

To reserve space for RC traffic in the schedule, we are going to minimize the makespan of the resulting MRCPSP-GPR. This will guarantee that at the end of each integration cycle there will be a gap without any TT traffic, and we will be maximizing the length of this gap.

Because, as we have discussed in section 3.3.2, this approach might not work very well in some cases, we also consider an alternative objective function. Instead of minimizing the makespan, we just limit the makespan by the length of the integration cycle, and we introduce *numGaps* gaps of equal length placed at regular intervals in the integration cycle. The gaps are modeled by tasks which require all the resources, i.e. no TT traffic can flow in the network during the gaps. Then, we will be maximizing the length of the gaps.

The problem of minimizing the makespan can be formally expressed using the `cumulativemm` and `genprecmm` constraints introduced in sections 4.2.3 and 4.2.4:

$$
\begin{aligned}
\min \ & \max_{j \in \mathcal{J}} (\boldsymbol{S}_j + \sum_{m \in \mathcal{M}_j} \boldsymbol{M}_{jm} \cdot \boldsymbol{p}_j) \\
\text{s.t.} \ & \texttt{cumulativemm}(\boldsymbol{S}, \boldsymbol{M}, \boldsymbol{p}, \boldsymbol{r}_{\bullet\bullet k}, \boldsymbol{R}_k) \ \ \forall k \in \mathcal{R} \\
& \texttt{genprecmm}(\boldsymbol{S}, \boldsymbol{M}, \boldsymbol{L}) \\
& \sum_{m \in \mathcal{M}_j} \boldsymbol{M}_{jm} = 1 \ \ \forall j \in \mathcal{J} \\
& \boldsymbol{M}_{jm} \in \{0, 1\} \ \ \forall j \in \mathcal{J}, m \in \mathcal{M}_j \\
& \boldsymbol{S}_j \geq 0 \ \ \forall j \in \mathcal{J}
\end{aligned}
$$

where $\mathcal{J}$ is the set of tasks, $\mathcal{M}_j$ is the set of modes of task j, $\mathcal{R}$ is the set of resources, $\boldsymbol{S}$ is the vector of the tasks' start times, $\boldsymbol{M}$ is a two-dimensional array of variables indicating if task j is executed in mode m, $\boldsymbol{p}$ a two-dimensional array of their processing times in individual modes, $\boldsymbol{r}$ a three-dimensional array of the resource requirements for each task and mode, $\boldsymbol{R}$ the vector of resource capacities, and $\boldsymbol{L}$ is a four-dimensional array of the lags between every two task-mode pairs.

To evaluate the resulting schedules of TT traffic, we are going to estimate the worst-case delays of RC traffic in the network. Unfortunately, we were not able to obtain benchmark data for the methods by [8], [9] and [3] outlined in section 3.2, and therefore we cannot make a direct comparison of our solutions with these. At least, we can compare the schedules obtained when using the two different objective functions mentioned above.

To avoid having to deal with routing the frames, we will assume a tree topology of the network so that there exists exactly one path between every two nodes. We believe that multiple possibilities of routing a frame could be expressed using the modes of task execution, but we leave this for future work.

It is also important to mention that we are using units of real time with arbitrary granularity when scheduling. This is a different approach from e.g. [8] where the time is divided into equal-sized slots, and it is assumed that each frame transmission occupies one slot. Using real time allows us to formulate the problem with more precision. However, as we will see later, it can also cause difficulties when solving the MRCPSP-GPR.

Chapter 6
Worst-case delay of RC traffic

Although we are not using the worst-case delay of RC traffic directly as a criterion we optimize for, we still want to be able to evaluate the quality of the obtained schedules from this point of view.

Originally, this was supposed to be a relatively minor part of this work. In the process, however, we found out that there are some issues with the state-of-the-art method presented in the literature. Therefore, we have decided to dedicate a chapter to this topic.

6.1 Calculating the worst-case delay

In [9], a method of estimating the worst-case delay based on the schedule porosity is presented. Using this method, the calculation is fast but it can be overly pessimistic. In [4], a new method is introduced which promises much tighter estimates of the worst-case delay. However, the improved precision comes at the cost of a much more expensive computation.

We are going to use the worst-case delay only to evaluate the quality of the solutions found. This means that the calculation is to be performed only once for each problem instance. Therefore, the performance is not such a concern as if the calculation was to be performed repeatedly during the solution process. The precision, on the other hand, is important for the evaluation of the solutions' quality.

Therefore, we attempted to use the more precise method by [4] which we describe below.

6.1.1 Sources of delay

Delays of RC frames can be caused by several factors. The following factors are considered:

- technical latencies
- queuing of RC frames
- scheduled TT frames
- TT and RC traffic integration

Technical latencies are introduced by the network nodes implementing various functionality like traffic policing and routing. It is assumed that these latencies are known in advance and that they are not frame-specific.

At the network switches the RC frames are received and routed for transmission on their target outgoing links. If multiple RC frames targetting the same outgoing link arrive within a short period of time, they need to be queued for transmission on this link, and the queuing delay is introduced. It is assumed that all frames have the same priority, and that the queue operates on the first in, first out (FIFO) basis.

Similarly, when the TT frames, which have higher priority, are being transmitted, the RC frames need to wait until the transmission of TT frames has finished. Therefore, the delay of the RC frames is increased by the time needed to transmit the TT frames.

With integration policy other than shuffling, the delay due to the TT traffic is further increased by the chosen integration policy. The timely-block policy is assumed to be used. Let l_{RCmax} be the maximum allowed size of a frame across all RC frames scheduled for transmission over the link. This policy mandates that an RC frame cannot be transmitted if there is not enough time to trasmit a frame of length l_{RCmax} before the next TT frame is scheduled. Therefore, there may be some idle intervals immediately preceding the transmission of TT frames on the data link.

6.1.2 Busy period

To estimate the maximum delay a RC frame f_x can be subjected to on a given dataflow link dl_j when arriving at a given time t_c^j, the concept of *busy period* is introduced. The busy period bp_x^j is defined as the time period between the instant when this RC frame f_x is queued for transmission on dl_j and the instant when all RC frames which had arrived before this frame have been transmitted. The transmission of the frame itself is also included in the busy period.

Before continuing further, we must note that we will deviate from the notation regarding the busy period used in [4] because we found it quite confusing. Instead, we will use an informal programming-like notation where e.g. $bp_x^j.length$ denotes the length of the busy period. Similarly, $f_x.BAG$ denotes the bandwidth allocation gap associated with the virtual link which the frame f_x belongs to.

According to [4], the length of the busy period can be expressed as

$$bp_x^j.length = Q_{dl_j}^{TT}\left(bp_x^j\right) + Q_{dl_j}^{RC}\left(bp_x^j, f_x\right) + Q_{dl_j}^{TL}\left(bp_x^j\right) + Q_{dl_j}^{TB}\left(bp_x^j\right) + C_x^j \qquad (1)$$

where $Q_{dl_j}^{TT}$, $Q_{dl_j}^{RC}$, $Q_{dl_j}^{TL}$, and $Q_{dl_j}^{TB}$ represent the times spent during the busy period transmitting the TT frames, the RC frames which had arrived before f_x, the sum of the technical latencies, and the idle time due to the timely-block policy, respectively. C_x^j denotes the transmission time of our frame f_x whose busy period is being analyzed.

Detailed explanation of the individual terms can be found in [4], here we will discuss the two most important terms only. $Q_{dl_j}^{TT}\left(bp_x^j\right)$ is equal to the sum of the transmission times of TT frames on the link within the busy period

$$Q_{dl_j}^{TT}\left(bp_x^j\right) = \sum_{\substack{f_i \in \mathcal{F}^{TT} \\ f_i \text{ in } bp_x^j}} C_i^j$$

$Q_{dl_j}^{RC}\left(bp_x^j, f_x\right)$ is the delay caused by the RC frames arriving before f_x

$$Q_{dl_j}^{RC}\left(bp_x^j, f_x\right) = \sum_{\substack{f_i \in \mathcal{F}^{RC} \\ f_i \neq f_x}} C_i^j \cdot \left\lceil \frac{bp_x^j.length}{f_i.BAG} \right\rceil$$

As the busy period appears in equation (1) on both sides, its calculation needs to be performed iteratively. Two concepts, *availability* and *demand* of the link are introduced for this purpose. The demand is the time needed to transmit all the RC frames including f_x

$$H_x^j\left(bp_x^j\right) = Q_{dl_j}^{RC}\left(bp_x^j, f_x\right) + C_x^j$$

and the availability is the time in the busy period available for the transmission of RC frames

$$A_x^j\left(bp_x^j\right) = bp_x^j.length - \left(Q_{dl_j}^{TT}\left(bp_x^j\right) + Q_{dl_j}^{TL}\left(bp_x^j\right) + Q_{dl_j}^{TB}\left(bp_x^j\right)\right)$$

When calculating the busy period starting at t_c^j, the algorithm starts with an initial length of bp_x^j equal to the sum of the lengths of all the RC frames. In each iteration the actual demand and availability are calculated and if the demand is not satisfied, the busy period is prolonged by this difference. This is repeated until the availability is greater than or equal to the demand. It is not discussed in [4] what the conditions for finiteness of this algorithm or alternative stopping criteria are. We can speculate that when deadlines for the RC frames are given, the busy period calculation is aborted once the busy period exceeds the frame's deadline.

It is a trivial observation that the length of the busy period depends on the time t_C^j when the RC frame arrives. Informally, if the frame arrives at a moment when the TT schedule is dense, the busy period will be longer than if it arrives when the TT schedule is less packed. Furthermore, if the BAGs of all the RC frames are large enough so that no RC frame is repeated during the busy period, it is easy to see that the end of the busy period $bp_x^j(t_c^j)$ is a non-decreasing function of the start t_c^j of the busy period. This follows from the fact that in such case the demand $H_x^j\left(bp_x^j\right)$ is independent of the busy period and that postponing the start time of a busy period while keeping its end fixed cannot increase the availability. In other words, a later start of the busy period cannot cause an earlier end of the busy period.

Unfortunately, this property does not hold once we consider shorter BAGs and the possibility of multiple instances of one RC frame within the busy period. For example, let's assume that we have two RC frames, f_1 with $BAG = 4$ and $L_max = 1$, and f_x with $BAG = 16$ and $L_max = 1$ whose busy period we want to calculate. In addition, the link is busy transmitting TT traffic in the interval $[0, 10)$. Then, if we calculate the busy period starting at 0 according to the above described algorithm, we obtain its end at 15 due to the increase in demand while waiting for link availability. In contrast, if we calculate the busy period starting at 10, it ends immediately at 12.

Furthermore, in such case the length of the busy period calculated according to this algorithm is no longer an upper bound on the actual delay of an RC frame arriving at t_C^j. Let's continue with the example above and consider f_x arriving at 10. There can be up to 3 queued instances of f_1 which arrived during the TT transmission block. Therefore, f_x will have to wait in the queue for 3 time units and its transmission will be finished at 14, i.e. after the end of the calculated busy period.

We will see further that this leads to some serious complications which eventually led us to discovering these less than desirable properties.

6.1.3 End-to-end delay

We could calculate the worst-case (across all frame arrival times) busy period for each dataflow link on a frame's path and sum these up to obtain an upper bound on the worst-case delay of the RC traffic. This could however lead to an overly pessimistic estimate as it is quite unlikely that after arriving at the most dense time and waiting for a long time in a queue at one dataflow link, the same situation would repeat itself at all other links. Therefore, the whole path of the frame is considered when calculating the end-to-end delay.

Like the busy period, the end-to-end delay is calculated for a single RC frame and a given point in time t_c^0 when the frame is dispatched for sending on the first link in its path. On each link, the frame's busy period is calculated and its end t^j is used as a base for the start t_c^{j+1} of the busy period on the next link. It would be possible to set $t_c^{j+1} = t^j$, however in [4] the time needed for transmission of the frames common to dl_j and dl_{j+1} is subtracted from t^j to obtain t_c^{j+1}. This refinement should prevent

duplicating the delays caused by traffic following the same path and thus improve the accuracy of the estimate. It is probably assumed that the delays on the first common link cause that on the following links the frames are spaced out enough not to interfere again.

The busy period calculation is repeated in this way until the final node of the path is reached. The resulting end-to-end delay estimate is the difference between the end of the busy period on the last link in the path and the start of the busy period on the first link in the path. If there are multiple paths within one virtual link, the maximum end-to-and delay for this virtual link is obtained by calculating the maximum across all the paths.

It can be proven that we can obtain an upper bound on the end-to-end delay this way if we assume that the length of the busy period is an upper bound on the maximum delay of the RC frame on the given link and that the end of the busy period is a non-decreasing function of its start. However we have seen that this is not necessarily true when the length of the busy period on some link is greater than the BAG of some RC frame. The example above shows this as a single dataflow link can be a special case of a virtual link.

6.1.4 Worst-case end-to-end delay

Given the algorithm for the calculation of the maximum delay of a frame sent out at t_C^j the authors of [4] state that the frame's worst-case end-to-end delay can be obtained by calculating the end-to-end delay for each time instant t_C^j in the schedule and selecting the greatest delay. Strictly speaking, this is impossible to do since there is a continuum of time instants within the application cycle.

Of course, we are working with discrete time so "each time instant" can be interpreted as each tick of the clock we are using, the elementary time unit. If we measure the time with a precision of nanoseconds, we would then need to calculate the end-to-end delay for every time instant from zero to application cycle length with a step of 1 nanosecond. As the application cycle is typically in the order of milliseconds to tens of milliseconds, this would require calculating the end-to-end delay for millions to tens of millions instances which is not feasible. If we used a more coarse unit we could reduce the number of samples but the calculation would still need a vast amount of computation. Actually, from the extremely long times needed to calculate the worst-case delays given by [4] it seems that this is the method which is used.

It is natural to think that if a slightly more pessimistic estimate of the worst-case delay was good enough, we could sample the time instants with a larger interval and speed up the computation. To obtain the estimate of the worst-case delay we would then add the length of the sampling interval to the calculated maximum. This is actually what we did to enable the calculation of the worst-case delay for problems with hunreds and thousands of frames in reasonable time.

What we observed was that selecting a shorter sampling interval sometimes led to a significantly higher worst-case delay estimate. This was quite unexpected and it made us realise that this approach was not feasible. The underlying problem lies in the assumption that the frame's calculated maximum arrival time to the last node is a non-decreasing function of its starting time on the first link. If it was so, we could take an interval $[t_1, t_2)$ where $\delta = t_2 - t_1$ is the length of the sampling interval. Then we would have t'_1, t'_1 the corresponding maximum arrival times. For the maximum arrival time t' of a frame starting at any $t \in [t_1, t_2)$ it would then hold

$$t' - t \leq t'_2 - t_1 = t'_2 - t_2 + t_2 - t_1 = (t'_2 - t_2) + \delta$$

The above assumption would hold if the end time of each busy period was a non-decreasing function of its start time, as we have already stated in 6.1.3. However, as we have shown above, this assumption only holds if the RC frames are not repeated within any busy period. Therefore, if we allow this repetition, we cannot guarantee that using this approach with sampling at longer intervals, we obtain a worst-case delay estimate greater than or equal to the estimate obtained considering each time instant.

6.1.5 Problem with the busy period

Further examination led us to believe that not only our modification with the sampling is at fault, rather there is a more fundamental problem with the method when allowing BAGs shorter than some busy period. In fact, such network and schedule can be constructed that even calculating the end-to-end delay for each time instant leads to an underestimation of the actual worst-case delay.

It might be argued that such large delaying of multiple RC frames resulting in their aggregation and sending in a burst not respecting the BAG is not allowed. However, we have not found an assumption in [4] ruling out this possibility and neither have we found information on how queing-related delays of RC frames are handled with respect to the BAG.

6.2 Repairing the worst-case delay calculation

It appears that the principal issue with the method introduced in [4] is how the busy period is calculated. At first sight it seems logical that if the busy period is longer than the BAG of some RC frame, multiple frame instances may be present in the queue and should therefore be included in the *demand* calculation. However, when looking more closely we realize that the frames received during the busy period are queued after the frame under analysis. Thus they should have no effect on the delay of this frame and, by definition, on the busy period.

On the other hand, it is neglected that there might be not only the BURST which arrived immediately before the analysed frame. There might also be other RC frames waiting which had been delayed e.g. by a long block of TT traffic preceding the BURST. Therefore, what we need to calculate is the maximum size which the backlog (the queue of RC frames waiting to be sent out over the link) can grow to. The time it takes to send out this amount of data starting at t_0 then gives us an upper bound on the delay of an RC frame arriving at t_0.

6.2.1 Dataflow link capacity requirements

Before presenting the method for calculating the maximum backlog size, we need to clarify the assumptions and requirements we need for the calculation to be meaningful. Intuitively, for the RC traffic to be schedulable, sufficient bandwidth has to be allocated for it. Otherwise the backlog could grow beyond all bounds.

To simplify the argument, let's assume for a moment that the RC traffic is continuous rather than discrete. The maximum bandwidth consumed by RC traffic on a given link can be approximated by

$$\widetilde{B}^{RC} = \sum_{f_i \in \mathcal{F}^{RC}} \frac{f_i.L_{max}}{f_i.BAG}$$

and the amount of data which may need to pass through the link during one application cycle is

$$\widetilde{Q}^{RC} = T_{appcycle} \cdot \sum_{f_i \in \mathcal{F}^{RC}} \frac{f_i.L_{max}}{f_i.BAG}$$

When calculating the link capacity available for RC traffic we need to consider both the time reserved for the TT traffic and the possibly idle time caused by the timely-block integration policy. In the worst case, there might be an idle period of l_{RCmax} during each of the blank intervals. (By blank intervals we mean the intervals between blocks of TT traffic on the link.) Therefore, the guaranteed capacity of the link available for RC traffic is

$$capacity^{RC} = \sum_{blank_l \in appcycle} \max(0, blank_l.length - l_{RCmax}) \tag{2}$$

Similarly, we define the guaranteed capacity within an interval $[t_0, t)$ by considering only the blank intervals (or their parts) within $[t_0, t)$

$$capacity^{RC}_{[t_0,t)} = \sum_{\substack{blank'_l = blank_l \cap [t_0,t) \\ blank_l \cap [t_0,t) \neq \emptyset}} \max(0, blank'_l.length - l_{RCmax}) \tag{3}$$

It would be natural to impose the constraint $\widetilde{Q}^{RC} \leq capacity^{RC}$ to ensure the link capacity is not exceeded. This could however lead to difficulty in proving the correctness of the maximum backlog calculation. Therefore, we define the extra capacity of the link

$$capacity^{extra} = capacity^{RC} - \widetilde{Q}^{RC}$$

and impose a stronger constraint

$$capacity^{extra} > 0$$

We will see that this constraint ensures that backlog of any size is eventually dissolved.

6.2.2 Maximum backlog size on a link

Let's assume that the first RC frame arives at time t_0. Then the backlog size BL at time $t \geq t_0$ can be calculated as

$$BL(t_0, t, Q_{in}) = Q_{in}(t_0, t) - Q_{out}(t_0, t, BL) \tag{4}$$

where $Q_{in}(t_0, t)$ is the amount of incoming RC traffic in the $[t_0, t)$ interval and $Q_{out}(t_0, t, BL)$ is the amount of data sent out over the link. Q_{out} depends on the size of the backlog, BL, because data can be sent out only when the size of the backlog is nonzero. Please note that we consider BL and Q_{out} to be higher-order functions, and their parameters Q_{in} and BL are functions, not just values.

To determine the maximum backlog size on the link we need to calculate

$$BL_{max} = \max_{t_0 \geq 0} \left(\max_{Q_{in}} \left(\max_{t \geq t_0} BL(t_0, t, Q_{in}) \right) \right) \tag{5}$$

where $\max_{Q_{in}}$ represents the maximum across all allowed incoming RC traffic patterns with the first frame coming at t_0 or later. Considering t_0 and the incoming RC traffic pattern separately may seem to be overly complicated but it will enable easier analysis.

First, we can notice that since the time-triggered traffic is periodic with the period of $T_{appcycle}$, the situation if the first RC frame arrives at $t_0 + T_{appcycle}$ is the same as if it arrives at t_0. Therefore, we can restrict ourselves to $t_0 \in [0, T_{appcycle})$.

Now we will show that we can consider only such t that $\forall t' \in [t_0, t) : BL(t_0, t', Q_{in}) > 0$ because it will not change the resulting BL_{max}. Let Q_{in} be any incoming RC traffic pattern with the first frame arriving at t_0 and let's assume that there is some time instant t' in the $[t_0, t)$ interval such that $BL(t_0, t', Q_{in}) = 0$. Let

$$\tilde{t} = \min\{t' \,|\, t_0 < t' < t \wedge BL(t_0, t', Q_{in}) = 0\}$$

Then $\forall t > \tilde{t}$

$$\begin{aligned} BL(t_0, t, Q_{in}) &= Q_{in}(t_0, t) - Q_{out}(t_0, t, BL) \\ &= Q_{in}(t_0, \tilde{t}) + Q_{in}(\tilde{t}, t) - Q_{out}(t_0, \tilde{t}, BL) - Q_{out}(\tilde{t}, t, BL) \\ &= \underbrace{BL(t_0, \tilde{t}, Q_{in})}_{=0} + Q_{in}(\tilde{t}, t) - Q_{out}(\tilde{t}, t, BL) \\ &= Q_{in}(\tilde{t}, t) - Q_{out}(\tilde{t}, t, BL) \\ &= BL(\tilde{t}, t, Q'_{in}) \\ &\leq \max_{Q_{in}} \left(\max_{t \geq \tilde{t}} BL(\tilde{t}, t, Q_{in}) \right) \end{aligned}$$

where Q'_{in} is an incoming RC traffic pattern identical to Q_{in} after $\tilde{t}$ and with no incoming RC traffic before this time instant. Therefore, when calculating BL_{max} we can limit the range of t to $[t_0, \tilde{t})$ where $\tilde{t}$ is the first time instant when the backlog size drops to zero.

Another question is whether the backlog size always drops to zero at some point in time. Let's assume that $\forall t' \in [t_0, t) : BL(t_0, t', Q_{in}) > 0$. Then the link is utilized for at least its guaranteed capacity so we can use the inequality

$$Q_{out}(t_0, t, BL) \geq capacity^{RC}_{[t_0, t)}$$

to remove the dependency of Q_{out} on BL from (4)

$$BL(t_0, t, Q_{in}) \leq Q_{in}(t_0, t) - capacity^{RC}_{[t_0, t)}$$

We can further remove the dependency on Q_{in} from the estimate

$$BL(t_0, t, Q_{in}) \leq Q^{max}_{in}(t - t_0) - capacity^{RC}_{[t_0, t)} \tag{6}$$

where $Q^{max}_{in}(t - t_0) = \max_{Q_{in}} Q_{in}(t_0, t)$. Due to the BAG and L_{max} limitations of each virtual link, we know that

$$Q^{max}_{in}(t - t_0) = \sum_{f_i \in \mathcal{F}^{RC}} \left(1 + \left\lfloor \frac{t - t_0}{f_i.BAG} \right\rfloor \right) \cdot f_i.L_{max}$$

We can estimate $Q^{max}_{in}(t - t_0)$ by removing the floor function to relate it to the link capacity requirements discussed in 6.2.1

$$Q^{max}_{in}(t - t_0) \leq \sum_{f_i \in \mathcal{F}^{RC}} \left(1 + \frac{t - t_0}{f_i.BAG} \right) \cdot f_i.L_{max}$$

For $t = t_0 + kT_{appcycle}$ where $k \in \mathbb{N}^+$ we then get

$$\begin{aligned} BL(t_0, t_0 + kT_{appcycle}, Q_{in}) &\leq Q_{in}^{max}(t - t_0) - capacity^{RC}_{[t_0, t_0 + kT_{appcycle})} \\ &= \sum_{f_i \in \mathcal{F}^{RC}} \left(1 + \frac{kT_{appcycle}}{f_i.BAG}\right) \cdot f_i.L_{max} - capacity^{RC}_{[t_0, t_0 + kT_{appcycle})} \\ &= \sum_{f_i \in \mathcal{F}^{RC}} f_i.L_{max} + k \cdot \underbrace{T_{appcycle} \sum_{f_i \in \mathcal{F}^{RC}} \frac{f_i.L_{max}}{f_i.BAG}}_{=\widetilde{Q}^{RC}} - k \cdot capacity^{RC} \\ &= \sum_{f_i \in \mathcal{F}^{RC}} f_i.L_{max} + k \left(\widetilde{Q}^{RC} - capacity^{RC}\right) \\ &= \sum_{f_i \in \mathcal{F}^{RC}} f_i.L_{max} - k \cdot capacity^{extra} \end{aligned}$$

and since we required that $capacity^{extra} > 0$, it is guaranteed that the backlog size drops to zero at most after $k = \lceil \frac{\sum_{f_i \in \mathcal{F}^{RC}} f_i.L_{max}}{capacity^{extra}} \rceil$ application cycles.

6.2.3 Estimating the maximum backlog size

Combining (5) with (6) and with the restrictions on the ranges of t_0 and t derived in the previous section, we obtain an estimate BL'_{max} of BL_{max}, which we will be able to compute

$$BL_{max} \leq BL'_{max} = \max_{0 \leq t_0 < T_{appcycle}} \max_{t_0 \leq t < \tilde{t}} \underbrace{\left(Q_{in}^{max}(t - t_0) - capacity^{RC}_{[t_0, t)}\right)}_{Q^{max}(t_0, t)} \tag{7}$$

where $\tilde{t}$ is the first time instant after t_0 when the backlog size drops to zero.

What remains to be determined is the sampling of t_0 and t for the calculation of the maximum backlog size. First we look at the sampling of t when t_0 is fixed. $Q_{in}^{max}(t - t_0)$ is a step-like increasing function when viewed as a function of t. It assumes the value $\sum_{f_i \in \mathcal{F}^{RC}} f_i.L_{max}$ at t_0 and then it may increase its value at time instants

$$t = t_0 + k \cdot \text{gcd}\{f_i.BAG \mid f_i \in \mathcal{F}^{RC}\} \tag{8}$$

As the subtraction of another increasing function $capacity^{RC}_{[t_0, t)}$ can never lead to an increase in the backlog size estimate, it is sufficient to sample t at these time instants for $k \in \mathbb{N}_0$.

Now we consider the sampling of t_0 with respect to the schedule of the TT traffic on the link. Let there be m blocks of TT traffic within one application cycle $[0, T_{appcycle})$. Let s_j be the start times of the j-th TT block, end_j its corresponding end, and $blank_j$ the non-occupied interval immediately preceding the TT block for $j \in \{1, 2, \ldots, m\}$. Furthermore, let's assume that $\forall j \in \{1, 2, \ldots, m\}$ the length of the blank interval $blank_j.length \geq l_{RCmax}$. Otherwise the interval would not be usable for RC traffic due to the timely-block traffic integration policy and we could merge the adjacent TT blocks into one larger TT block for the purpose of the calculation.

We will show that it is enough to sample t_0 at time instants

$$t_0^{(j)} = (s_j - l_{RCmax}) \tag{9}$$

If $(s_1 - l_{RCmax}) < 0$ we need to take the sample at $(s_1 - l_{RCmax} + T_{appcycle})$ instead.

First we show that for Q^{max} defined in (7) and $\forall j, \forall t_0 < t_0^{(j)}, t_0 \in blank_j$

$$\max_{t_0 \le t < \tilde{t}} Q^{max}(t_0, t) \le \max_{t_0^{(j)} \le t < \tilde{t}^{(j)}} Q^{max}(t_0^{(j)}, t) \tag{10}$$

Let $\delta > 0$ and $t_0 = t_0^{(j)} - \delta$ such that $t_0 \in blank_j$, and examine the difference

$$\begin{aligned} Q^{max}(t_0^{(j)}, t) - Q^{max}(t_0, t-\delta) &= Q^{max}(t_0^{(j)}, t) - Q^{max}(t_0^{(j)} - \delta, t - \delta) \\ &= Q_{in}^{max}(t - t_0^{(j)}) - capacity^{RC}_{[t_0^{(j)}, t)} - \\ &\quad - Q_{in}^{max}(t - \delta - t_0^{(j)} + \delta) + capacity^{RC}_{[t_0^{(j)} - \delta, t-\delta)} \\ &= capacity^{RC}_{[t_0^{(j)} - \delta, t-\delta)} - capacity^{RC}_{[t_0^{(j)}, t)} \\ &\ge 0 \end{aligned}$$

The last inequality is valid thanks to our choice of $t_0^{(j)}$ and δ. When the start of the interval is shifted by δ to the left, the length of the first blank within this interval increases from l_{RCmax} to $(l_{RCmax} + \delta)$, and therefore its contribution to the interval capacity increases by δ. At the same time the capacity of any interval decreases at most by δ when shifting its end by δ to the left. Together, the shift of the interval by δ to the left cannot decrease its capacity, and the inequality holds $\forall t$ and therefore for the maxima as well.

Similarly we show that (10) holds $\forall t_0 > t_0^{(j)}, t_0 \le end_j$ too. Using δ analogously, we obtain

$$\begin{aligned} Q^{max}(t_0^{(j)}, t) - Q^{max}(t_0, t+\delta) &= Q^{max}(t_0^{(j)}, t) - Q^{max}(t_0^{(j)} - \delta, t - \delta) \\ &= capacity^{RC}_{[t_0^{(j)} + \delta, t+\delta)} - capacity^{RC}_{[t_0^{(j)}, t)} \\ &\ge 0 \end{aligned}$$

and the argument for the validity of the inequality is only slightly different. When shifting the start of the interval by δ to the right no capacity is lost because the first blank was making no contribution to the capacity, and the same holds for the following TT block. Of course, when shifting the end of any interval to the right, its capacity cannot decrease. Together, the inequality holds $\forall t$ and therefore for the maxima as well.

To summarize, to find an upper bound BL'_{max} on the maximum backlog size for a link at any time, we need to sample $Q^{max}(t_0, t)$, defined in (7), at points $(t_0^{(j)}, t^{(k)})$ where $t_0^{(j)}$ and $t^{(k)}$ are defined in (9) and (8).

6.2.4 Worst-case delay on a link

Once we know the upper bound BL'_{max} on the maximum backlog size on the link, we can easily estimate the maximum delay of any RC frame on this link when it arrives at some t_0. Any frame is added to the backlog at the moment of its arrival, so at t_0 the frame is already part of the backlog. As we are interested in the worst-case delay, we assume that the frame is at the last position in the backlog, i.e. that the whole backlog has to be sent out for the frame to be sent further. Therefore, the maximum delay of any RC frame can be estimated by the time needed to send out BL'_{max} when starting at t_0.

The time needed to send out BL'_{max} when starting at t_0 can be calculated in a very similar way to how the busy period in the original method is calculated. The only difference is that the *demand* is fixed to BL'_{max} instead of being updated in every iteration. For faster calculation, some minimum step size for prolonging the busy period can be set at the cost of worsening the estimate by at most the step size. Alternatively, instead of simple incrementing the end time by the difference between the demand and availability, we can look at the schedule and fast-forward to the next blank period where the availability can increase.

Using a very similar argument to that presented in section 6.1.2 it can be shown that the end time of sending out the maximum backlog is a non-decreasing function of t_0 when the sending started.

6.2.5 Worst-case end-to-end delay

To calculate the worst-case end-to-end delay of a frame on a given link we can use a very similar method to that described in sections 6.1.3 and 6.1.4. The only difference we make is choosing the start times on the next link to be equal to the end times on the previous link without any optimizations, i.e. $t_c^{j+1} = t^j$ (using the notation of [4] like in those sections).

Thanks to the nice property of t^j being a non-decreasing function of t_c^j we can also do the sampling of t_c^0 proposed in 6.1.4. All the reasoning made in 6.1.3 and 6.1.4 can be applied to show that using this method we obtain a valid upper bound on the worst-case end-to-end delay of an RC frame over a selected path in a virtual link. To estimate the worst-case delay across all RC traffic in the network, we just need to find the maximum of the upper bounds across all virtual links in the network and across all paths in each virtual link.

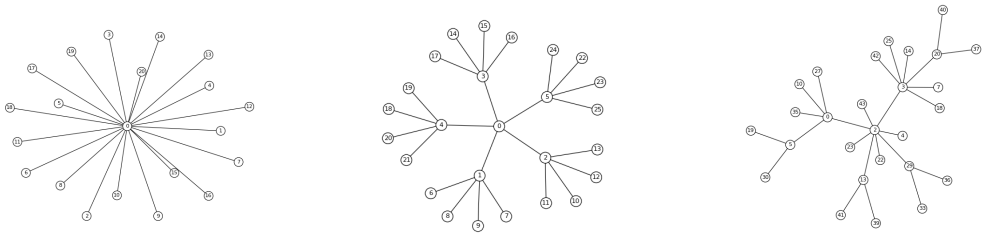
Chapter 7
Experimental results

7.1 Test instance generator

Because we had no real-world data or standard benchmark data available for evaluating the scheduling methods, we have created a test instance generator. The generator is based on the description of test instances used by [8], and it can generate the application-level constraints as described in section 3.1.2.

7.1.1 Parameters

Network topology. In [8] three network topologies are proposed for benchmarking, a star, snowflake and a random tree. Generating a star and a snowflake is straightforward, for the random tree we need to choose the method of generating it. We chose making use of the Barabási-Albert model which is available through the `barabasi_albert_graph` function of the NetworkX library. To avoid having long linear segments in the network, which are uninteresting for the scheduling, we apply edge contraction to eliminate all nodes of degree 2. For all the topologies, we generate networks with 20 leaf nodes, i.e. 20 end systems. The topologies which can be generated are depicted in fig. 7.1.



star snowflake random tree

Figure 7.1. Supported topologies.

Frames. To simulate different classes of end devices, partitioning the set of leaf nodes into three groups, broadcast, multicast and unicast, which have different rules on where they can send the frames, is proposed by [8] where more details can be found. We have implemented this partitioning in the generator but we have not used it for testing. Instead, we allow that any leaf node can send frames to an arbitrary subset of other leaf nodes, which is also one of the configurations considered in [8].

To allow comparison with [8], we will use frames of uniform size (the minimum size of an Ethernet frame) and uniform periods as one test case. For a more realistic evaluation and for assessment of how additional complexity affects the difficulty of the problem, we will use random frame sizes and periods as another test case. To keep the length of the application cycle reasonable, we want the frame periods to be harmonic or as close to harmonic as possible. We will therefore limit ourselves to periods of $\tau = 2^n 3^m \tau_{base}$ where $m \in \{0, 1\}$, $n \in \mathbb{N}$ and τ_{base} is some base period.

Application-level constraints. There is already a limit on the end-to-end delay of the TT frames implied by the requirement that the transmission has to be done within one integration cycle. We will not impose additional constraints on the end-to-end delays.

To simulate the task interdependence constraints, we partition the set of frames into groups of size 5 (with the last group possibly having a smaller size), and among the frames within each group we create a random precedence tree with minimum lags specified. We let the lags to be in the order of 10 % of the integration cycle length.

7.1.2 Implementation

We have implemented the test instance generator in the Python 3 programming language. We have selected Python because it allows rapid development, and mainly because the NetworkX[29] library is available for Python. The NetworkX library allows us to generate the graphs and to work with them easily, e.g. when constructing the virtual links for the messages.

7.1.3 Test instance format

We use XML as the output format for the generator. We made this choice because we wanted the format to be easily human-readable, which we later really appreciated when debugging. Because our model has been under dynamic development, we have not yet formalized the format using an XML schema. As the format is very verbose, it is quite easy to understand it from the example instance, which is part of the attached source codes. Therefore, we will only discuss it briefly. An illustrating example is given in Appendix A.

The `node` and `link` elements define the network's topology (one `link` element for each direction of a duplex physical link), and the `message` elements define the parameters of each frame and its sender and consumers. The task interdependence constraints are represented by the `app_group` elements and the `precedence` elements inside them. Because the routing of the frames can be considered a part of the schedule, it is defined under the `schedule` element. As a virtual link is a tree, it is also represented by a tree in the XML, and each link in the virtual node is stored as a child of the link preceding it. Frames are assigned to virtual links using the `msgsched` elements. For easier processing and better human-readability, each `msgsched` element contains `schedentry` elements, one for each link in the virtual link, although these could be seen as redundant. Since the routing of TT frames follows the same rules as the routing of RC frames, we use the `vlink` elements for routing the TT frames as well.

Because we have limited ourselves to finding the schedule when routing of the frames is given, we generate all the elements listed above using the instance generator. When the schedule of the TT frames is known, an `offset` is added to each of the frames' `schedentry` elements, specifying the offset of the transmission within the application cycle. Such XML file with the schedule specified can then be used as an input for calculating the worst-case delay.

7.2 Other implemented tools

Unlike when generating the test instances, performance can play an important role when calculating the worst-case delays of RC traffic, as suggested by the computation times reported by [4]. Therefore, we have chosen C++ for its implementation. It implements the method proposed by [4] modified as described in section 6.2.

It addition to the XML-based network description including the offsets of the TT frames, it takes the required precision as an input, and its result is a single number, the estimate of the worst-case delay across all RC frames in the network.

Before running any of the solvers, we need to transform the XML-based network description into a problem description which the solvers will understand.

As a part of the code of the SCIP-based solver provided to us by Dr. Schnell, we received some utility code for handling input instances in the ProGen/max input format (defined at [15]). This utility code reads the problem instance and stores it in a data structure which makes it easy to formulate the model for the actual solver. We have partially refactored this code and adapted it to allow formulating a model for the CP Optimizer as well.

However, we still need to transform the network description into the MRCPSP-GPR in the ProGen/max format. For this, we have created another utility program which implements the transformation to MRCPSP-GPR as outlined in chapter 5, and prints the resulting problem in the required format.

7.3 Solver performance

To explore the capabilities and limits of the solvers we are using, we have generated several test instances with the number of TT frames ranging from 20 to 2000. We have selected the snowflake topology, and for each sender node, a random subset of the other nodes is selected as receivers. This leads to a total number of frame instances (transmissions of a frame over individual links) ranging approximately from 120 to 12000.

With the exception of the instances with 20 frames, where this led to infeasibility, we set the length of the integration cycle so that the minimum possible load of the busiest link in the network in its busiest integration cycle is approximately 50 %. This minimum load is also a lower bound on the total makespan of the schedule, and we obtain it using CPLEX to solve a simple MIP model which we use for the load balancing heuristic as described in section 3.3.3.

We performed the tests on a x86_64 GNU/Linux with the 3.13.0 kernel, Intel i5-3320M CPU and 8 GB of DDR3 RAM, and we set a limit of 300 s on the computing time. The SCIP-based solver runs in a single thread while the CP Optimizer uses one thread for each CPU logical core, i.e. 4 threads on our machine, unless set otherwise. As we wanted to evaluate the best readily achievable performance of the solvers, we have kept this automatic setting.

We have generated a set of test instances in which all the frames have the same size, and their period is equal to the length of the integration cycle. This simplifies the problem because in such case, each task can be executed in a single mode only, and the same sizes effectively allow dividing the timeline into slots of fixed length.

First, we compared the performance of the CP Optimizer and the SCIP-based solver when minimizing the makespan of the schedule. The results are given in Table 7.1. To clarify the symbols used, n denotes the number of frames, N the corresponding number of frame instances, l_{cycle} the length of the integration cycle, and LB is the lower bound on the makespan. The asterisk indicates that the time limit was reached. The solution time is given in seconds, the other values representing time are given in nanoseconds (the time unit is unimportant for the comparison, though).

				CP Optimizer		SCIP-based	
n	N	l_{cycle}	LB	makespan	time	makespan	time
20	128	20000	5376	14728	0.03	14728	0.08
50	309	25000	14784	18677	78.99	19938	300*
100	619	50000	28224	30240	2.03	49951	300*
200	1217	100000	51072	53088	46.88	82239	300*
500	3064	250000	125664	132051	300*	250000	300*
1000	6119	500000	246624	255301	300*	387225	300*
2000	12325	1000000	487200	499474	300*	N/A	N/A

Table 7.1. Comparison of the solvers.

We can see that using the CP Optimizer, we are able to find either the optimal solution for the minimum makespan or a solution which is close ($< 10\%$ gap) to the lower bound within 300 seconds. The SCIP-based solver is able to find the optimal solution for the smallest instance only, and the solutions it finds for the larger instances within the time limit are significantly worse than those produced by the CP Optimizer. Furthermore, the SCIP-based solver does not perform well when working with large numbers. We have observed during the solution process, long runs of decreasing the objective function by 1 at a time appear, and they can consume a large part of the computation time. With the largest test instance, the SCIP-based solver failed to run because of problems with integer overflow (this is, however a feature of the implementation of the `genprecmm` constraint handler rather than SCIP itself). For these reasons, we have excluded the SCIP-based solver from further experiments.

To be fair, however, we have to mention that the performance of the SCIP-based solver depends significantly on an initial upper bound given to the solver. It often finds a solution just slightly better than the given upper bound and then it struggles improving it, taking a much longer time to obtain some better solution than if a smaller initial upper bound is provided to it. In this case, we gave the solver the length of the integration cycle as the upper bound on the makespan, and we can see that for the instance with 500 frames, it found a solution with exactly this makespan. Therefore, it might be possible to improve its performance through some scheme attempting to run the solver with varying value of the upper bound. Some degree of parallelism could also be achieved this way without having to parallelize the solver itself. (This idea is not original, it has been presented in [16] and to some extent implemented in the code of the solver.)

Next, for the same test instances, we have tried to insert 5 equally spaced gaps into the integration cycle and to maximize their length using the CP Optimizer. The results are given in Table 7.2. Instead of the lower bound on the makespan, we now have an upper bound on the length of one gap $gapUB = \left\lfloor \frac{l_{cycle} - LB}{5} \right\rfloor$.

We can see that attempting to introduce the gaps into the schedule is much more difficult for the CP Optimizer than just minimizing the schedule. Only the test instances with up to 200 frames can be reasonably scheduled, then the performance deteriorates rapidly. We will evaluate this method with respect to the worst-case RC delay, however, we will exclude it from further performance evaluation.

To assess the performance of the solver more realistically, we have generated another set of test instances with random frame sizes and periods. We let the frame payload

n	N	l_{cycle}	$gapUB$	gap length	time
20	128	20000	2924	856	1.17
50	309	25000	2043	968	139.15
100	619	50000	4355	3280	300*
200	1217	100000	9785	5968	300*
500	3064	250000	24867	1290	300*
1000	6119	500000	50675	440	300*
2000	12325	1000000	102560	8	300*

Table 7.2. Maximizing the length of multiple gaps.

be uniformly distributed in the $[1, 256]$ interval and their periods be 1, 2, or 3 times the length of the integration cycle, again, uniformly distributed. This introduces the modes of task execution, which make the problem much more difficult to solve.

We evaluate the performance of the CP Optimizer on the multi-mode problem, and compare it to applying the load-balancing heuristic for mode selection and then solving a single-mode problem. The results are given in Table 7.3. For the heuristic, the time needed for the load-balancing is also given.

				multi-mode		heuristic	
n	N	l_{cycle}	LB	makespan	time	makespan	time
20	127	50000	11952	30710	0.42	30710	0.01+0.08
50	331	50000	32776	40907	300*	40413	0.03+300*
100	615	100000	52808	56921	300*	57774	0.03+300*
200	1288	200000	112944	131123	300*	127328	0.09+300*
500	3158	500000	278968	301300	300*	299902	0.21+300*
1000	6195	1000000	515424	549934	300*	548982	0.43+300*
2000	12460	2000000	1040270	N/A	300*	1077810	0.98+300*

Table 7.3. Influence of the heuristic.

We can see that the variable frame sizes and periods introduce extra complexity compared to the uniform frame sizes and periods. Now, the CP Optimizer can find the optimal solution and prove its optimality within the time limit for the smallest instance only, even when the multi-mode complexity is eliminated by the heuristic. Still, the solutions are reasonably close to the lower bounds on the makespan.

The load-balancing heuristic leads to a slight increase of the solution quality in most cases. This does not, however, tell us the whole story. Naturally, it can happen that using the heuristic can lead to finding a worse solution, as it happened in the case of the test instance wih 100 frames. On the other hand, the heuristic enabled the solver to find a very good solution for the instance with 2000 frames, which the solver was unable to find by itself. We also observed that using the heuristic led the solver to find decent solutions much earlier in the solution process.

7.4 Worst-case RC delays

In addition to evaluating the performance of the solvers, we wanted to assess the suitability of the two objective functions – minimizing the makespan or introducing multiple gaps into the schedule and maximizing their length – with respect to minimizing the worst-case delays of RC traffic.

For this assessment, we need a good solution with respect to each of the objective functions, and therefore, we need to choose the instance size accordingly. We will use test instances with 100 TT frames, all with their period equal to the integration cycle length and their size equal to the minimum size of an Ethernet frame, and an integration cycle of length 50000 ns. As we have seen, this produces approximately 50 % load of the busiest resource. For modeling RC traffic, we use frames with payload (representing L_{max}) uniformly distributed in the $[1, 256]$ interval and their BAG randomly selected from $\{2, 4, 8\}$ times the length of the integration cycle. We will use 40 such frames. This should lead to an additional load of approximately 40 % for the link most loaded with RC traffic. Together, we should get a network with a relatively high load, but which is not overloaded (approximately 90 % load if the maximum loads concentrate on the same link).

We generate 5 test instances on which we compare the worst-case delays of RC traffic when optimizing the schedule for the minimum makespan or for maximum gap lengths (with 5 gaps in the integration cycle). The results are given in Table 7.4.

	worst-case delay of RC traffic (ns)	
test instance	makespan minimized	gap length maximized
A	130633	141600
B	133880	141223
C	150230	165888
D	125128	145258
E	155720	155536

Table 7.4. Worst-case delays of RC traffic.

We can see that for our test instances with high load of the network and a relatively large volume of RC traffic, maximizing the length of the gaps does not produce better results than minimizing the makespan of the schedule. Combined with the fact that maximizing the length of the gaps is computationally much more demanding, we suggest abandoning this objective function and focusing on minimizing the makespan. An exception to this might be a situation if we had only a small volume of RC traffic and we required very small delays. In such case, a long block of TT traffic could itself cause an excessive delay, and therefore the gaps would be needed.

Chapter 8
Conclusion

In this work, we have explored TTEthernet and the existing approaches to scheduling TTEthernet traffic. Then, we have presented a transformation of the problem of scheduling TTEthernet traffic into the multi-mode RCPSP with generalized precedence relations (MRCPSP-GPR), and we have proposed two objective functions which could lead to creating schedules favorable for rate-constrained (RC) traffic. Because the MRCPSP-GPR is very difficult to solve, we have presented a simple load-balancing heuristic for selecting and fixing the execution modes of individual tasks, and thereby simplifying the problem to single-mode RCPSP-GPR.

To enable assessing the practicality of our approach, we have implemented a generator of test instances because no real-world data were available to us. We have used the generated test instances to evaluate the two objective functions, the heuristic, and the performance of available MRCPSP-GPR solvers. We have measured the performance of two solvers, a solver based on the SCIP framework and the IBM CP Optimizer, of which the CP Optimizer appears to perform better by a large margin. When combined with the load-balancing heuristic, the CP Optimizer is able to schedule up to thousands of frames, which is close to the size of practical problems.

Apart from scheduling the TT traffic, which was the main focus of this work, we have discovered a flaw in a published state-of-the-art method for estimating the worst-case delay of RC traffic in a TTEthernet network. We have been able to repair the method and to present a semi-formal proof of its correctness. We have then implemented the repaired method and used it to evaluate the suitability of the two proposed objective functions with respect to minimizing the worst-case delays of RC traffic.

Future work could focus on removing some additional constraints we have imposed on our schedule, especially the requirement that every TT frame has to be transmitted end-to-end within one integration cycle. Further, routing the frames in more complex network topologies could be considered. If more information on TTEthernet becomes available, it is also important to review all the constraints and the test instance generator in order to align them with the requirements of real-world applications.

References

[1] B. Galloway and G. P. Hancke. Introduction to Industrial Control Networks. *IEEE Communications Surveys Tutorials*, 15(2):860–880, Second Quarter 2013.

[2] *AS6802: Time-Triggered Ethernet*. SAE International, 2011.

[3] Domitian Tamas-Selicean, Paul Pop, and Wilfried Steiner. Synthesis of Communication Schedules for TTEthernet-based Mixed-criticality Systems. In *Proceedings of the Eighth IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*, CODES+ISSS '12, pages 473–482, New York, NY, USA, 2012. ACM.

[4] D. Tamas-Selicean, P. Pop, and W. Steiner. Timing Analysis of Rate Constrained Traffic for the TTEthernet Communication Protocol. In *2015 IEEE 18th International Symposium on Real-Time Distributed Computing*, pages 119–126, April 2015.

[5] *ARINC 664P7: Aircraft Data Network, Part 7, Avionics Full-Duplex Switched Ethernet Network*. ARINC (Aeronautical Radio, Inc.), 2009.

[6] *AFDX / ARINC 664 Tutorial (1500-049)*. Condor Engineering, Inc., 2005.

[7] *Time-Triggered Ethernet – A Powerful Network Solution for Multiple Purpose*. TTTech Computertechnik AG.

[8] W. Steiner. An Evaluation of SMT-Based Schedule Synthesis for Time-Triggered Multi-hop Networks. In *Real-Time Systems Symposium (RTSS), 2010 IEEE 31st*, pages 375–384, Nov 2010.

[9] W. Steiner. Synthesis of static communication schedules for mixed-criticality systems. In *Object/Component/Service-Oriented Real-Time Distributed Computing Workshops (ISORCW), 2011 14th IEEE International Symposium on*, pages 11–18, March 2011.

[10] W. Steiner, M. Gutiérrez, Z. Matyas, F. Pozo, and G. Rodriguez-Navas. Current techniques, trends and new horizons in avionics networks configuration. In *2015 IEEE/AIAA 34th Digital Avionics Systems Conference (DASC)*, pages 1–26, Sept 2015.

[11] Domitian Tamas-Selicean, Paul Pop, and Wilfried Steiner. Design optimization of TTEthernet-based distributed real-time systems. *Real-Time Systems*, 51(1):1–35, 2015.

[12] S. S. Craciunas, R. S. Oliver, and V. Ecker. Optimal static scheduling of real-time tasks on distributed time-triggered networked systems. In *Proceedings of the 2014 IEEE Emerging Technology and Factory Automation (ETFA)*, pages 1–8, Sept 2014.

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

[14] D. Waraus Z. Hanzálek, D. Benes. Time constrained FlexRay static segment scheduling. In *Proceedings of the 10th International Workshop on Real-Time Networks*, pages 61–67, Porto, Portugal, 2011.

[15] Multi mode project duration problem MRCPSP/max.
http://www.wiwi.tu-clausthal.de/en/abteilungen/produktion/forschung/schwerpunkte/project-generator/mrcpspmax/.

[16] Alexander Schnell and Richard F. Hartl. On the efficient modeling and solution of the multi-mode resource-constrained project scheduling problem with generalized precedence relations. *OR Spectrum*, 38(2):283–303, 2016.

[17] Tobias Achterberg. SCIP: Solving constraint integer programs. *Mathematical Programming Computation*, 1(1):1–41, 2009.
http://mpc.zib.de/index.php/MPC/article/view/4.

[18] IBM CPLEX CP Optimizer, 2015.
http://www-01.ibm.com/software/commerce/optimization/cplex-cp-optimizer/.

[19] Timo Berthold, Stefan Heinz, Marco E. Lübbecke, Rolf H. Möhring, and Jens Schulz. *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems: 7th International Conference, CPAIOR 2010, Bologna, Italy, June 14-18, 2010. Proceedings*, chapter A Constraint Integer Programming Approach for Resource-Constrained Project Scheduling, pages 313–317. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010.

[20] Tobias Achterberg. Conflict Analysis in Mixed Integer Programming. *Discret. Optim.*, 4(1):4–20, March 2007.

[21] Tobias Achterberg. *Constraint Integer Programming.* PhD thesis, Technische Universität Berlin, 2007.
http://opus.kobv.de/tuberlin/volltexte/2007/1611/.

[22] ZIB ACADEMIC LICENSE.
http://www.zib.de/berthold/ZIBopt/academic.txt.

[23] Tobias Achterberg, Timo Berthold, Thorsten Koch, and Kati Wolter. *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems: 5th International Conference, CPAIOR 2008 Paris, France, May 20-23, 2008 Proceedings*, chapter Constraint Integer Programming: A New Approach to Integrate CP and MIP, pages 6–20. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008.

[24] G. Hendel, A. Gleixner, and F. Serrano. Implementing Constraint Handlers in SCIP, 2015.
http://co-at-work.zib.de/files/20150930-Hendel-slides.pdf.

[25] SCIP User's Manual.
http://scip.zib.de/doc/html/.

[26] CO@Work 2015 – Combinatorial Optimization at Work.
http://co-at-work.zib.de/.

[27] Alexander Schnell and Richard F. Hartl. On the generalization of constraint programming and boolean satisfiability solving techniques to schedule a resource-constrained project consisting of multi-mode jobs. *Working paper (under review)*, 2015.

[28] Petr Vilím, Philippe Laborie, and Paul Shaw. *Integration of AI and OR Techniques in Constraint Programming: 12th International Conference, CPAIOR 2015,*

Barcelona, Spain, May 18-22, 2015, Proceedings, chapter Failure-Directed Search for Constraint-Based Scheduling, pages 437–453. Springer International Publishing, Cham, 2015.

[29] Aric A. Hagberg, Daniel A. Schult, and Pieter J. Swart. Exploring network structure, dynamics, and function using NetworkX. In *Proceedings of the 7th Python in Science Conference (SciPy2008)*, pages 11–15, Pasadena, CA USA, August 2008.

Appendix A
Example XML-based instance description

To illustrate the structure of the XML-based instance description we have used, an example is shown below. Please note that it is an extremely reduced example, and therefore, it is not a valid network description. A more complete example can be found in `SourceCodes/RCdelay/test.xml` on the attached CD.

```
<network>
    <settings>
        <baseperiod>1000000</baseperiod>
    </settings>
    <node>
        <name>Node1</name>
        <type>end</type>
        <techdelay>0.0000025</techdelay>
    </node>
    <link>
        <startNode>Node1</startNode>
        <endNode>Switch1</endNode>
        <bandwidth>1e9</bandwidth>
        <techdelay>0.000005</techdelay>
    </link>
    <message>
        <name>TTMsg1</name>
        <type>TT</type>
        <payload>12</payload>
        <period>3</period>
        <sender>Node1</sender>
        <consumers>
            <consumer>Node2</consumer>
            <consumer>Node3</consumer>
        </consumers>
    </message>
    <app_group>
        <precedence>
            <pred>TTMsg1</pred>
            <succ>TTMsg2</succ>
            <minLag>861</minLag>
        </precedence>
    </app_group>
    <schedule>
        <vlink>
            <name>VLink1</name>
            <link>
                <name>Node1Switch1</name>
                <link>
                    <name>Switch1Node2</name>
```

```
                </link>
                <link>
                    <name>Switch1Node3</name>
                </link>
            </link>
        </vlink>
        <msgsched>
            <msgname>TTMsg1</msgname>
            <msgvlink>VLink1</msgvlink>
            <schedentry>
                <linkname>Node1Switch1</linkname>
                <offset>0.0</offset>
            </schedentry>
        </msgsched>
    </schedule>
</network>
```

Appendix B
Content of the attached CD

- `Heller_thesis.pdf` – a digital version of this work
- `SourceCodes/`
  - `MRCPSPNew_src/` – source code of the MRCPSP-GPR solvers
  - `RCdelay/` – source code for the transformation of the XML-based network description into MRCPSP-GPR and for RC traffic delay calculation
  - `TTEInstanceGenerator/` – source code for generating the test instances and for interpreting the MRCPSP-GPR solvers' output

Each of the source code directories contains a `README` file providing basic information about the code.