Universidade Federal da Paraíba

Centro de Informática

Programa de Pós-Graduação em Informática

TSNSCHED: Automated Schedule Generation for Time Sensitive Networking

Aellison Cassimiro Teixeira dos Santos

Dissertação submetida à Coordenação do Curso de Pós-Graduação em Informática da Universidade Federal da Paraíba como parte dos requisitos necessários para obtenção do grau de Mestre em Informática.

Área de Concentração: Ciência da Computação

Linha de Pesquisa: Satisfiability Module Theory | Scheduling | Time Sensitive Networking

Vivek Nigam

Iguatemi Eduardo da Fonseca

(Orientadores)

João Pessoa, Paraíba, Brasil

Resumo

Time Sensitive Networking (TSN) é um conjunto de padrões que habilitam alta performance e comunicação determinística utilizando escalonamento de tráfego. Devido ao tamanho das redes industriais, configurar redes TSN manualmente é uma tarefa desafiadora. Nós apresentamos TSNsched, uma ferramenta para geração automática de cronogramas para redes TSN. TSNsched recebe como entrada a topologia lógica da rede, expressada por meio de fluxos, e retorna cronogramas para os switches TSN fazendo uso de um resolvedor SMT. O cronograma gerado garante a performance desejada para a rede (especificada em forma de latência e variação da latência), considerando que tal cronograma existe. TSNsched suporta a especificação de fluxos unicast e multicast, como em sistemas seguindo o padrão Publish/Subscribe; é capaz de combinar variações existentes do problema de escalonamento de tráfego em redes TSN, e considera tanto tráfego de melhor esforço quanto tráfego prioritário para estas redes. TSNsched pode ser usado como uma ferramenta independente e também permite rápida prototipação por meio de sua API em JAVA. Nós avaliamos TSNsched em um conjunto de topologias de tamanho semelhante a cenários reais. TSNsched é capaz de gerar cronogramas de alta performance, com latência média de menos de 1000μs, e variação da latência média de menos de 20μs, para redes TSN com até 138 assinantes e 10 fluxos multicast.

Palavras-chave: TSN, SMT, escalonamento.

Abstract

Time Sensitive Networking (TSN) is a set of standards enabling high performance deterministic communication using time scheduling. Due to the size of industrial networks, configuring TSN networks is challenging to be done manually. We present TSNsched, a tool for automatic generation of schedules for TSN. TSNsched takes as input the logical topology of a network, expressed as flows, and outputs schedules for TSN switches by using an SMT-solver. The generated schedule guarantees the desired network performance (specified in terms of latency and jitter), if such schedules exist. TSNsched supports unicast and multicast flows, such as, in Publish/Subscribe networks; can combine existing variants of TSN scheduling problems, and reason about the best-effort and priority TSN traffic. TSNsched can be run as a standalone tool and also allows rapid prototyping with the available JAVA API. We evaluate TSNsched on a number of realistic-size network topologies. TSNsched can generate high performance schedules, with average latency less than 1000μs, and average jitter less than 20μs, for TSN networks, with up to 138 subscribers and up to 10 multicast flows.

Keywords: TSN, SMT, Scheduling

Acknowledgements

Aos meus pais Lúcia Maria Teixeira Cassimiro dos Santos e Anacleto Cassimiro dos Santos, por terem construído o caminho que me trouxe aqui hoje. Vivi sob a luz de pessoas humildes, e que deram a mim muito mais que qualquer um neste mundo poderia oferecer.

Ao meu irmão Allison, por me dar seu exemplo de esforço e resiliência. Tudo se consegue ao persistir diante das adversidades do mundo.

À Bianca Amorim, pelo companheirismo, amizade, apoio, experiências e bons momentos. Acompanhar de perto alguém crescer e florescer com tanto ímpeto, mesmo cercada por obstáculos, muitas vezes me motivou a dar passos para frente quando minhas próprias forças falharam. Se sou alguém melhor, muito a devo por isto.

Ao meu colega, amigo e orientador Vivek Nigam, por ter sido meu pai acadêmico. Sete anos atrás, em uma conversa tão simples, tive a chance de ver através dele a perspectiva de um mundo imenso que parecia tão distante. Hoje posso dizer que trilhei parte da estrada na qual nunca me imaginei dando um passo.

Ao meu colega, amigo e orientador Iguatemi Eduardo da Fonseca, por ter formado a pessoa e profissional que sou. Sou eternamente grato pelo suporte, a confiança, as oportunidades que me fizeram crescer e, acima de tudo, o conhecimento vindo de suas experiências.

To Ben Schneider, for your knowledge and support. I am glad that it was given me the chance for us to work together and hope that the future holds many more years of collaboration. Vielen Dank!

To Abraão Aíres, Anand Subramanian, Cesare Tinelli, Haniel Barbosa, Kristin Yvonne Rozier and Nestan Tsiskaridze, for the time, attention and consideration. Each of them was kind enough to share life experiences and perspectives of great importance to me. Few words can truly change a life.

Contents

List of Symbols

CoS : *Class of Service*

CDT : *Control-Data Traffic*

FIFO : *First In First Out*

GCD : *Greatest Common Divisor*

GCL : *Gate Control List*

GRASP : *Greedy Randomized Adaptive Search Procedure*

IIoT : *Industrial Internet of Things*

ILP : *Integer Linear Programming*

LCM : *Least Common Multiple*

MIP : *Mixed Integer Programming*

NW-JSP : *No-Wait Job Scheduling Problem*

PCP : *Priority Code Point*

SAT : *Satisfiability*

SMT : *Satisfiability Module Theories*

TAS : *Time-Aware Shaper*

TDMA : *Time-Division Multiple Access*

TSN : *Time-Sensitive Networking*

TSSDN : *Time-Sensitive Software-Defined Networks*

TT : *Time-Triggered*

UBS : *Urgency-Based Scheduler*

UDAP : *User-Defined Application Period*

List of Figures

List of Tables

Source Code List

Chapter 1

Introduction

As observed in the later years of technological advancements, there is an increasing demand for faster, precisely coordinated, automated systems. We see such behavior, for instance, on the fields of industrial machinery, aerospace, smart cities and many others crucial areas for the functioning of modern society. On these fields, the constant need for improvements on high communication speed, reliability and determinism led to the development of the novel technology titled Time-Sensitive Networking (TSN), offering these needed advantages to standard IEEE 802.1 and IEEE 802.3 Ethernet networks [18].

In-vehicle network applications can be mentioned as examples of systems with deterministic requirements, which already use TSN for some of its operations with plans for expansions [25; 17]. It is vital that applications such as driver-assistance have their packets delivered with speed and reliability, having in mind the consequences that can be generated by the failure of this system. In the literature, there are analysis on such application using TSN technology with three different methods of traffic scheduling, simulating a real life scenario, showing that latencies of up to 106 microseconds and maximum average jitter of 26 microseconds can be obtained [31]. Still, there are instances where these values hardly satisfy the requirements of the system to which it was designed for.

Simply put, the TSN technology can assure determinism by controlling the moment of transmission for certain packets at each hop of the communication path. One of the available approaches to do so is by assigning classes to streams of packets, and at each hop of the path there is a specific time window where packets from a specific class can be transmitted. This time window is repeated after a fixed amount of time, creating a cyclic behavior for the transmission of the packets. Discovering the size of these cycles, the starting time and duration of these time windows, and assigning the classes to the streams such that the time taken to travel from the source to the destination is lower than a user-specified value, as well as the variation of this time for every packet of a stream, are the core of the scheduling problem.

For relatively small systems (*e.g.*, a few streams and a switch), this scheduling problem can easily be solved manually, but when taking bigger scenarios in consideration where more devices are interconnected, requirements are stricter, and domain specific knowledge about the requirements of the streams is needed, such approaches for solving this problem are not feasible. For this reason, automated approaches for generating schedules and assigning classes to streams are more suitable to the task.

Still, to develop such approaches implies in facing challenges born from the complexity of the scheduling problem, as the major problem for TSN to be successfully adopted in the industry is the complex configuration of the TSN schedules, given that the problem is NP-complete [8].

While there are multiple other protocols, technologies and standards available for covering scenarios in the Industry 4.0 and other systems with similar requirements (e.g. Profinet, EtherCAT, Ethernet/IP), many fail to be combined with standard Ethernet networks and devices, and adapting these systems implies in the usage of tailored solutions [32].

As it is being currently developed, the TSN technology can be seen as a composite of other standards, each designed to cover a set of responsibilities which, when integrated, provide the necessary grounds for the implementation of deterministic communication.

From these standards, the one that can be considered the core of the time-triggered communication paradigm and also further explored in this dissertation is the IEEE 802.1Qbv, responsible for the management of queues and traffic in switches on TSN networks. Its implementation works as the Time-Aware Shaper (TAS) of a switch; a module responsible for handling the opening and closing of gates of egress packet queues.

The IEEE 802.1Qci standard also plays an important part in the scheduling approach discussed here, as it provides the necessary grounds to identification of streams and modification of packet headers performed on switches [7]. As it will be later discussed, although it is not necessary, this functionality can improve the efficiency of TSNsched for generating schedules.

As later explained on Chapter 2, TSN switches work under a time multiplexed system where specific time slots are used inside a cycle to manage the gates of the queues in the egress ports. A schedule will specify when to open and close the gates. The schedule must take in consideration the constraints of the *flows* (a concept of streams adapted to the TSN scenario) and the network itself, such as maximum latency allowed, maximum jitter of the flow, the existence of a time space between transmission time slots, so on and so forth.

When arriving at a switch, a packet will be directed to an egress queue according to the class assigned to the flow. We also use the terminology *priority* to refer to the class of the stream, and *priority queue* to refer to the egress queue in which packets of a specific class (or, in this case, priority) are placed.

At the conception of this technology, the task of finding the timing variables of a schedule for a priority queue's gate opening and closing as well as the assignment of priorities, was manually solved, but it can be an error prone activity.

The difficulty of the problem can be escalated when considering the management of multicast Publish/Subscribe flows, adding more constraints to the system as a single flow sends packets in a multicast model. Similarly, other approaches to improve the efficiency of the schedules generated, such as reservation of bandwidth for best-effort traffic, can hinder the process of obtaining an answer to the scheduling problem.

1.1 Objectives

1.1.1 General Objectives

To develop a tool capable of scheduling traffic for TSN networks implementing both unicast and multicast flows.

1.1.2 Specific Objectives

- To specify a formal model of the TSN scheduling problem;
- To develop a tool capable of automatically generating schedules for TSN networks using SMT solvers;

- To develop a tool capable of generating arbitrary networks modeled according to the architecture of TSN networks, aiming to benchmark the scheduling tool;
- To evaluate the tool and validate its scalability.

1.2 Research proposal

The formulation of a schedule for TSN networks is an NP-complete problem [7], and it can be challenging to generate schedules that comply with the system constraints. While there are multiple techniques for handling scheduling problems, the complexity for generating a general solution and necessary time to present a schedule play an important part on deciding which tools and methods will be used on this research.

Having this in mind, we propose a scheduler for TSN compliant networks that solves the scheduling problem using an SMT solver.

Thangamuthu *et al.* discusses three different shapers for TSN traffic in in-vehicle switched applications, showing that, without further restrictions for specific use cases, the shapers hardly satisfy the requirements for the studied scenarios [31]. In the case of the TAS, the scheduling method can make all the difference in the end-system. This highlights the importance of having a generic tool capable of generating schedules without specific knowledge on the scenario.

As SMT solvers are renowned for their capabilities of bringing solutions to complex problems (including in scheduling problems [34]), it has been decided that a good approach for investigation would be a modeling of the scheduler using the Z3 solver. Also, the flexibility offered by the usage of first order logic to specify the problem are one of the advantageous features and important for the task. We highlight that the approach presented in this document is not dependant on the tool Z3, as other solvers could be used with the same purpose.

The tools and methods available in the state of the art [22][8][10][15], aside from being proprietary, have no control over the fine tuning of the schedule quality, bandwidth reservation for best-effort traffic and flexibility of the schedule, as in allowing that packets arrive outside of their transmission window, which are some of the points covered in this research. Also, unicast and multicast flows, specifically in a Publish/Subscribe model, are discussed in this work.

Using JAVA to model the network and the Z3 API to specify its constraints, this dissertation presents an automated schedule generation tool for TSN networks titled TSNSCHED.

By breaking a flow into multiple flow fragments, we create a more modular, maintainable implementation and help the understandability of the constraints. It also allows us to express more conditions and properties, when compared to other encodings such as [6; 30; 7]. This expressiveness can be used to quickly add and remove constraints to the scheduling problem, which creates a flexible tool for generating schedules for TSN.

Moreover, as there are no openly available TSN flow benchmarks, a tool that creates multicast and unicast flows in a given network topology was created. This tool was used to generate a set of benchmarks, containing flows of different sizes, to evaluate TSNSCHED. This benchmark can be used to evaluate and compare other future TSN schedule generation tools.

1.3 Dissertation Structure

In the second chapter we lay and explore the necessary grounds for discussing about topics on Time-Sensitive Networking. Since the general basis on TSN was given in this chapter, a deeper look into the Time-Aware Shaper will be taken, as well as the modeling of a network and the foundations of the Z3 SMT solver.

We discuss the tools, algorithms and models available in the state of the art for scheduling traffic in TSN networks, as well as some other insightful related works, in the third chapter. A special light will be cast to the ones also based on SMT solvers.

In the fourth chapter, we describe in depth the schedule generation process of TSNSCHED and the formal model implemented by the tool presented in this dissertation.

In the fifth chapter, we analyze and discuss the implementation of TSNSCHED, the topology generator, it is experiments and results.

The sixth chapter contains final considerations about the work of this dissertation.

Chapter 2

Theoretical Framework

The objective of this chapter is to present the basis of knowledge used to build TSNSCHED. Here, the concepts of networking needed to comprehend the scheduling problem for Time-Aware Shapers, the control of the gates of TSN switches and topics on Z3 and SMT solvers will be discussed.

2.1 Time-Sensitive Networking

Time Sensitive Networking (TSN) is a recent OSI layer 2 network protocol standard (IEEE 802.1Q [20]) that extends Ethernet. It is developed to address the increasing performance demands of, for example, industrial automation applications like motion control, where the movement of components has to be synchronized in a microsecond range [18]. Many of these applications require multicast flows (*i.e.*, sequences or, using another technical term, streams of packets) as the same data (*e.g.*, position of the axis) is sent to different actuators, as well as combining high priority traffic with best-effort traffic (*e.g.*, video streaming).

Being part of Industry 4.0, TSN enables efficient communication between numerous small disconnected networks, building a unified network structure. Of the multiple applications of TSN in the Industry 4.0, prime examples of it are seen in: factory automation, as the network convergence allows for efficient interaction of large machinery and systems with a large number of robots; energy automation, as the determinism offered by TSN allows the transmission of time-critical data, such as sampled values from voltage and current on electrical substations; and transportation applications, as the reservation of bandwidth for priority traffic allows for applications such as control functions and passenger entertainment to share a network [2].

Interacting components of these network structures have timing requirements for their communication, as a delay in a message might cause synchronization problems in sub-parts of the system represented by the network. The guarantee in determinism, which handles these requirements, offered by TSN can be seen in the form of controlled latency and jitter. In simple terms, latency is the time taken for a packet to travel from its source node to its destination node, and the jitter of a packet is the variation of its latency from the average latency of the packet's stream.

Table 2.1: TSN IEEE substandards according to [4; 32; 21]

	Standard Title	**Description**
802.1Qbv	Enhancements for Scheduled Traffic	Deterministic scheduling of traffic in queues through switched networks
802.1ASrev, IEEE 1588	Timing and Synchronization for Time-Sensitive Applications	Synchronization clocks of the nodes in the network and fault tolerance management
802.1CB	Frame Replication and Elimination for Reliability	Handling the redundancy of frames throughout the network
802.1Qca	Path control and reservation	Static path reservation, control and configuration
802.1Qbu and 802.3br	Frame Preemption	Handling of frame preemption (Interrupting and resuming frame transmission)
802.1Qcc	Enhancements and performance improvements	Allows enhancements and performance improvements Central configuration method
802.1Qch	Cyclic Queuing and Forwarding	Specifies synchronized cyclic enqueuing and queue draining proceduresk
802.1Qci	Per-Stream Filtering and Policing	Perform frame counting, filtering, policing, and service class selection for a frame

TSN achieves these high performance requirements by a set of sub-standards which provide, for example, high precision time synchronization and different schedulers for time-triggered packet forwarding [20; 19]. The list of standards and a brief description of its purpose is shown in Table 2.1. Moreover, TSN allows the co-existence of best-effort flows, as in traditional networks, and flows that take advantage of the high precision time synchronization (*e.g.* flows that demand high performance).

Of these standards, we must highlight the ones that provide the necessary operations and concepts used by TSNSCHED.

The IEEE 802.1ASrev and IEEE 1588 standards, responsible for the time synchronization of nodes in the network, allows systems to give the same concept of time to all switches and devices. Assuming that all nodes work under the same clock, we are now able to give precise values for the latency and jitter of packets.

The IEEE 802.1Qbv allows systems to use a Time Division Multiple Access (TDMA) policy for handling the transmission of packets, controlling the gates of egress queues in ports according to a schedule. This allows packets to be transmitted with priority over other sorts of traffic. This mechanism is also entitled the Time-Aware Shaper, and it is elaborated in Section 2.2.

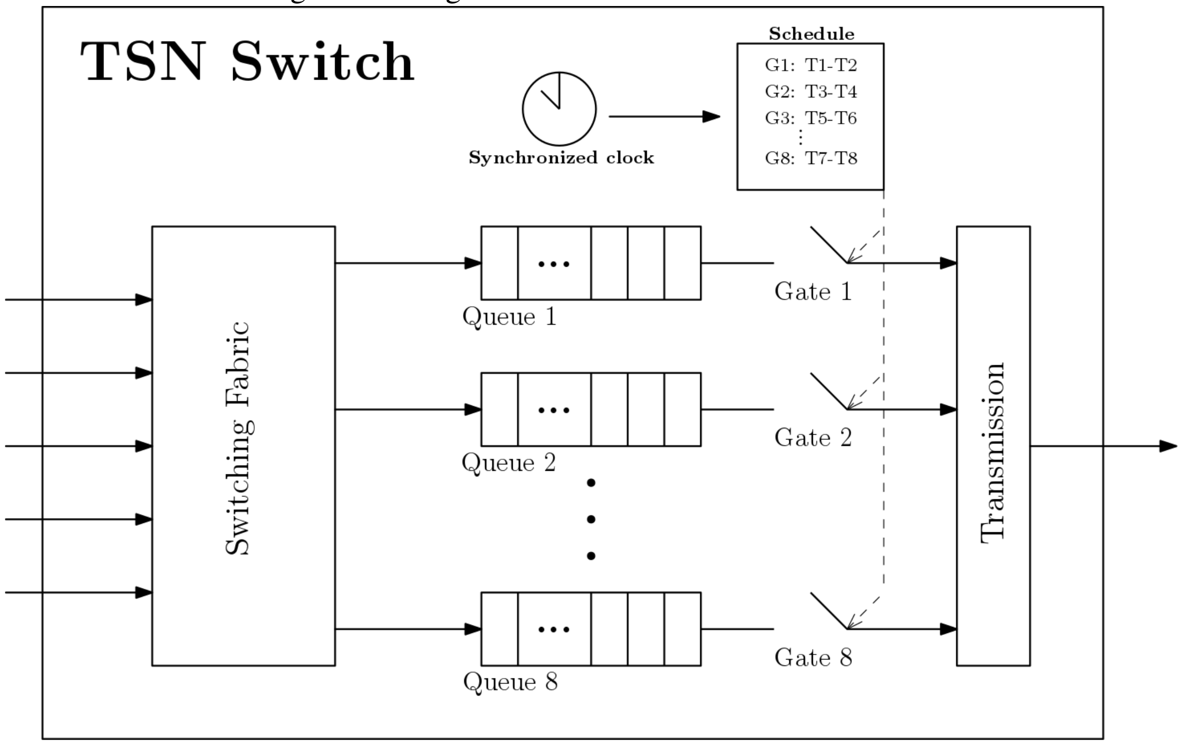
Lastly, the IEEE 802.1Qci gives switches the ability to filter packets and to select classes for a frame (a message from a node to another, or simply a packet, as we mostly refer). In other words, a packet can have different priorities in different switches, allowing for a more dynamic approach for scheduling transmission windows. This is done by identifying packets on a flow (the terminology for stream in TSN) level and modifying the packet header. Although the approach presented in this dissertation can also be used in networks where switches do not implement this standard, we can see that its usage greatly improves the execution time of TSNSCHED, as shown in Section 5.3.

2.2 Time-Aware Shaper

As mentioned in the previous chapter, the Time-Aware Shaper (TAS) is one of the core mechanisms of TSN. The control of the gates, as well as the placing of the packets in the proper queues is decisive to guarantee the delivery of packets respecting the flow's constraints.

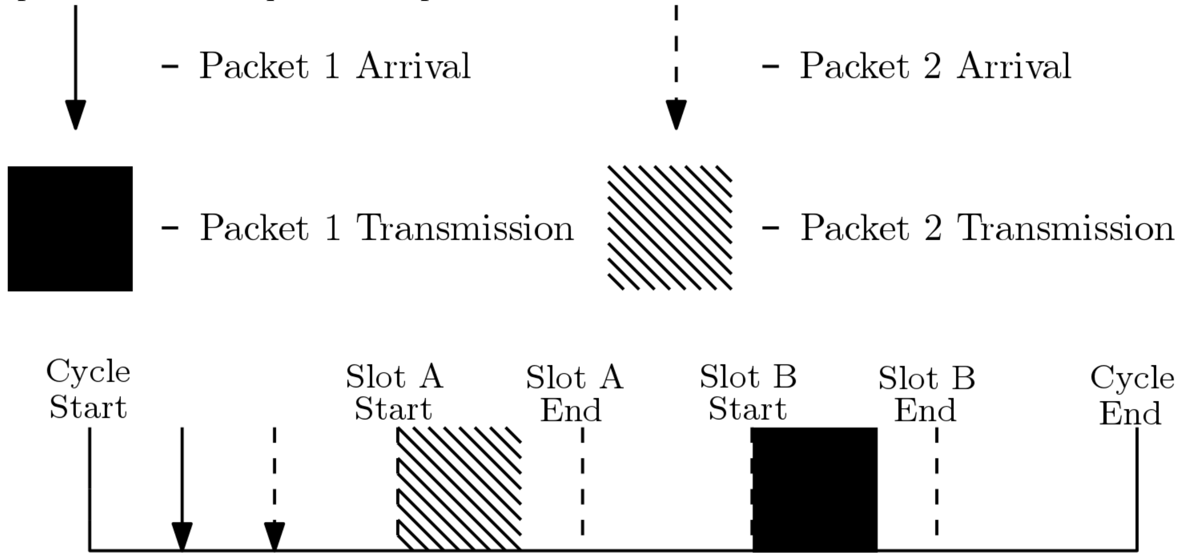
The Figure 2.1 can be used to better understand how the gate management system works. After synchronizing the clock of the devices in the network, the packets received by the switch are filtered by a switching fabric in order to identify the egress port used to reach the packet's destination. To specify the priority queue used by the packet, the Class of Service (CoS) (represented by 3 bits of the VLAN tag of the Ethernet header, hence, there can be up to 8 priorities) is used to implement a Priority Code Point (PCP) system, which, in turn, will be used to direct the packet to the proper priority queue of the egress port [18]. By doing so, it is possible to classify traffic by priorities and separate it from best-effort (*i.e.*, non-critical) traffic. The queues will operate under a First In First Out (FIFO) policy, where the packets will only be able to leave once the gate of the queue is open. The main point of the determinism given by TSN is the schedule contained in a Gate Control List (GCL), which is responsible for opening and closing the gates of the queues according to the schedule. That way, the schedules given to the GCLs of the switches can be created having in mind the constraints of the devices and the system.

Figure 2.1: High level abstraction of a TSN switch.



The schedule is specified by a cycle containing time slots to each of the priorities in use. Each priority can have multiple time slots. Also, there must be a time space where no queue gates are open between two non-consecutive slots, called guard band, and is used to avoid packet collision since there might be an attempt of transmission right before the beginning or at the end of a time slot (with not enough time left to transmit).

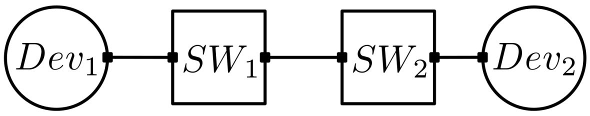
Figure 2.2: A cycle, its time slots and the transmission of packets of different priorities (placed in different queues) in a port of a TSN switch.



In a temporal perspective, an example of the handling of packets in a switch can be seen in Figure 2.2. In this figure, we see that time intervals within the cycle are used to transmit packets belonging to priority traffic. Given that Packet 1 and Packet 2 are assigned to priority queues A and B, respectively, Packet 1 can only be transmitted once the gate of its priority queue is open, *i.e.*, within the span of the transmission window for priority A. The same is valid to Packet 2 and its priority queue. It is important to highlight that packets of different priorities can be transmitted out of order, once they are placed in different queues, which is the case shown in Figure 2.2.

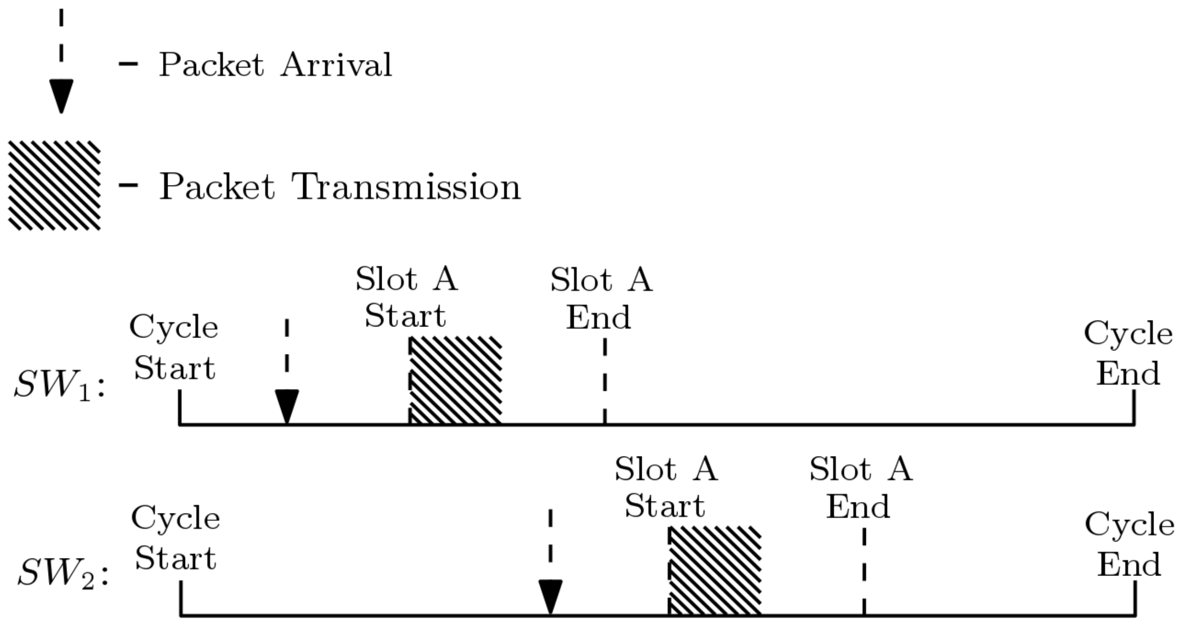
With this level of control, schedules can be built in ways where the delay of the packets is minimized and the jitter is controlled. To exemplify this statement, lets consider a simple network composed of two devices and two switches, as shown in figure 2.3. A flow of packets goes from the sender device Dev_1 to the receiver device Dev_2, traveling through two switches SW_1 and SW_2. We can manipulate how much time the packets take to reach Dev_2, given that the Dev_1 sends data on a time-triggered policy with fixed periodicity, by creating slots of time where only the packets from this flow can be transmitted in the two switches.

Figure 2.3: A topology with two switches and two devices.



Still considering the example shown in Figure 2.3, we see in Figure 2.4 how time windows can be created in order to assist in the guarantee of determinism. Assuming that packets belonging to the stream from Dev_1 to Dev_2 have been assigned to the priority i that uses the transmission slot A, Dev_1 sends a packet to SW_1, which waits for the beginning of the time slot A to close all open gates of the egress queues of that port and open the gate for the priority queue i. In the case shown in the figure, we also assume that SW_1 and SW_2 have the same cycle start and duration, but this might not be the case in other scenarios. Packets from this stream can be transmitted while the gate of the priority queue i is open, *i.e.*, throughout the course of the duration of the time slot A, in order to reach SW_2. This process is repeated in SW_2 for sending packets to Dev_2.

Figure 2.4: Transmission of packets in a temporal perspective.



Since we consider the usage of devices that periodically send data on a time-triggered policy, we know the moment when these packets leave the sender device, and knowing when they can be transmitted in each switch of the path gives us enough information to know how long it takes for them to reach their destination. That said, for simple scenarios such as the one we just discussed, it is possible to adjust the duration of a cycle to be equals to the periodicity of the packet sending of the sender device, and the transmission window for these packets can be configured to start right at their arrival. This exemplifies a scenario where the latency of the packets is minimal, consisting only of the time taken to be transmitted and to travel between nodes, since the packet is immediately forwarded at each hop, and jitter is non-existent, as the duration of the cycle is the same of the periodicity, a packet will be sent at the same moment every cycle.

2.3 The Scheduling Problem

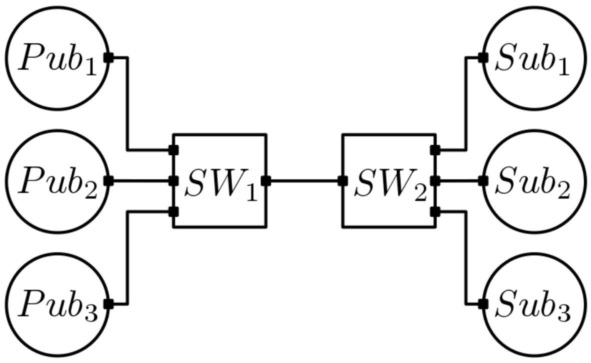
With the TAS, the problem now is to assign proper schedules to the GCLs. These schedules must be built in a manner where the transmission of the packets in each hop respect the constraints of the stream, which means that, for example, if a network has a flow with ten switches in its path, the schedule given to the first switch must take in consideration the time taken to traverse the other nine switches. The more switches shared in the path of flows, the more complex it becomes to create the schedule. The problem also becomes more complex if the number of flows sharing switches in their paths increase.

The priority of the flow is also another incognito variable of the problem, and must be discovered in order to make sure that packets from a flow will use the correct transmission window. If only the implementation of the IEEE 802.1Qbv standard is used, it is assumed in the scheduling problem that a flow is assigned to a priority, meaning that at every hop of the path, the packet will be placed at the same priority queue. Still, if the system is also IEEE 802.1Qci compliant, then we can assume that a switch can identify the stream to which a packet belongs and override its PCP in order to place it on a queue of different priority. In other words, if we are designing a schedule for a system where its switches implement the IEEE 802.1Qbv and IEEE 802.1Qci, then every packet from a flow can have a different priority in every switch of the path (priority change per hop).

It is important to emphasize that this identification happens on a stream level, meaning that two different packets from the same flow cannot have different priorities on the same switch.

A concrete example of a (trivial) scheduling problem is shown in Figures 2.5 and the correlating schedule in Figure 2.6.

Figure 2.5: Network topology with Publishers (Pub) and Subscribers (Sub) connected via a single link (SW_1 to SW_2)

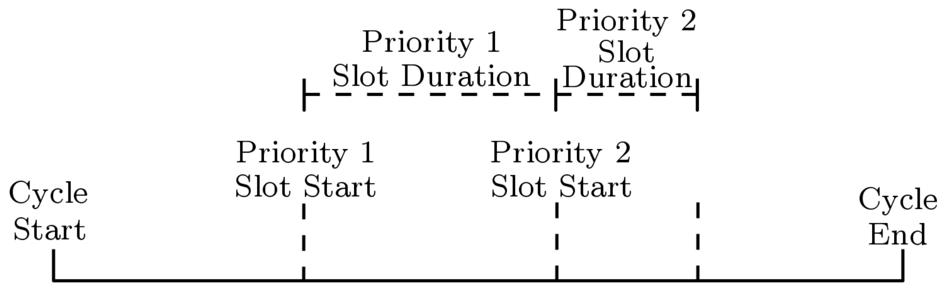


Follwing a Publish/Subscribe policy, the example topology contains Publishers Pub and Subscribers Sub and two switches SW_1 and SW_2. Lets assume that $Flow_1$, from Pub_1 to Sub_1, belongs to a safety critical application with hard real-time requirements sending periodic traffic. $Flow_2$, Pub_2 to Sub_2, periodically transmits process data into the cloud for analysis purposes and has deterministic requirements. $Flow_3$, Pub_3 to Sub_3, is a best effort stream which requires high bandwidth (*e.g.*, video data for visualization) but is not critical for the function of the process or the safety of the plant. In this setup all streams compete for bandwidth on the link between SW_1 and SW_2.

Without a proper scheduling mechanism it might happen that the low priority best effort $Flow_3$ consumes the bandwidth which is needed by the higher prioritized $Flow_1$ and $Flow_2$. TSN enables the scheduling of flows by reserving bandwidth in a time-triggered schedule. According to 802.1Qbv [20], the TAS which reserves bandwidth by assigning different flows to their appropriate priority (1-8) and finding the appropriate time slot for each of the priority queues on a transmission port. The example TAS schedule shown in Figure 2.6 shows a valid configuration where $Flow_1$ is mapped to priority p_1, $Flow_2$ is mapped to priority p_2 and the rest of the cycle is open for transmitting best effort traffic.

The scheduling problem of the TAS can be summarized as finding the cycle duration and the points in time for opening and closing the time slots, that is, the time slot starting time and time slot duration [30]. Moreover, these slots shall specify some specific TSN requirements, *e.g.*, be separated by a guard band under some conditions.

Figure 2.6: IEEE802.1Qbv schedule for the port that connects SW_1 to SW_2



This time-triggered scheduling problem is NP-complete [6] and solving it manually is a time-consuming and error-prone task that needs detailed domain-specific knowledge about the communication system and application.

Moreover, current trends show that the complexity of scheduling will grow in the future. An example can be given from the industrial automation domain where topologies continue to grow (known as IIoT – Industrial Internet of Things). Therefore, the requirements of future industrial networks make manual configuration infeasible.

Multiple methods of schedule building and priority assignment have and are being developed and explored in the state of the art, as it will be discussed in the Chapter 3, but still, attributes such as average latency, jitter, feasibility and execution time of the scheduler are topics to be discussed when choosing a scheduling method.

2.4 Z3 SMT solver

When proving theorems and solving problems, the complexity of a question might be a great hindrance to the creation of a forward proof, *i.e.*, starting from axioms and assumption and searching for a proof. One of the ways of dealing with such a problem is, once a goal is obtained, work backwards from the goal respecting the properties of the problem in order to map sub-goals and find the ones capable of proving the given theorem [23]. Such approach can simplify the definition of a theorem solver; a tool where, given constraints and a goal, fills in the gaps in order to build an answer.

Aiming at problems derived from software verification and software analysis, Z3 is a renowned Satisfiability Module Theory (SMT) solver capable of efficiently handling multiple data types and formulas [11].

Model generation is also done by Z3, which means that the assignment of values to constants given as input is possible. With a generated model, the user can query the values of the goal obtained by the tool and easily formulate a concrete scenario that solves a given problem.

To exemplify the usage of Z3 and how it can be used to generate a schedule for TSN networks, we can observe a few simple problems solved by the tool. The first example is a simple theorem expressed as follows:

$$\begin{aligned} &x, y \in Z. \\ &x + y = 100 \end{aligned} \tag{2.1}$$

The problem here lies in the definition of values for x and y. For the discovery of these values, a short specification of the problem can be written following the SMT-LIB standard (for usage in different SMT solvers) as shown in the Code Excerpt 2.1. To do so, two constants representing x and y must be created and a assertion implying that the sum of both is equal to 100 must be stated. After doing so, it is possible to check if the problem is solvable and, if it is, retrieve the desired values from the generated model.

Source Code 2.1: Simple theorem definition in Z3.

```
; Simple value search

(declare-const x Int) ; Creating an integer constant x
(declare-const y Int) ; Creating an integer constant y

(assert (= (+ x y) 100)) ; x + y = 100

(check-sat) ; Check if it is solvable
(get-model) ; Retreive model if solvable

(eval x) ; Print x value
(eval y) ; Print y value
```

Since no other conditions over the values were implied here, Z3 returns the first set of values that can be used as a solution to the specified problem (which in this case was *x = 100* and *y = 0*). Constraints can be added to the problem in the form of new assertions. The following problem shows how the theorem can become slightly more complex:

$$\begin{aligned} & x, y \in Z; 40 \leq x \leq 60; \mid x - y \mid \geq 5. \\ & x + y = 100 \end{aligned} \tag{2.2}$$

The new constraints to the problem are added as shown in Code Excerpt 2.2.

Source Code 2.2: Simple theorem definition with further assertions in Z3.

```
(declare-const x Int)
(declare-const y Int)

(assert (= (+ x y) 100))
(assert (>= x 40))
(assert (<= x 60))
(assert
  (or
    (>= (- x y) 5)
    (>= (- y x) 5)
  )
)

(check-sat)
(get-model)

(eval x)
(eval y)
```

The new assertions added to the code create further boundaries to the values that x and y can assume (the model now returns *x = 47* and *y = 53*). Now, not only x must be inside a fixed interval, the space between x and y must be of at least a certain value. This can allude to a simple example of how the transmission of a packet can be specified, as one packet must be transmitted inside a time window and there must be a interval of at least the transmission time between the transmission of two packets.

In TSNSCHED, the difficulty of the problem is much greater due to the amount of assertions involving boundaries to multiple time slots, priority assignment, network and flows constraints.

Z3 is also a multi-plataform tool [33]. While the standalone Z3 code can be used for solving scheduling problems, the whole problem must be manually defined before querying the solver for answers. For this reason, the Z3 Java API version 4.8.0.0 was used during the development of this dissertation, since the creation of dynamic use cases becomes a much simpler process.

2.5 Closing Remarks

With the discussion of the TAS, the reader is now able to understand the problem of scheduling packets and why it is important to be solved. Also, the composition of the network can help the understanding of what should be taken in consideration when modeling a topology, as well as how complex the problem can become when scaled.

The introduction to Z3 also covers the basics of problem specification and the idea behind the discovery of values. While the problems further discussed are much harder to model and solve, the idea behind the attainment of a solution remains the same.

Being aware of this points, the user now possesses the necessary basis to understand TSNSCHED.

Chapter 3

Related Work

As the TSN technology is still in its early years in the market, with vendors applying their own implementations to fill certain gaps in the standards specifications [4], the material available in the state of the art is relatively new.

Still, due to the importance of the TAS for TSN networks, the amount of work available for analysis in the state of the art of this specific field is resourceful. In this chapter, we will discuss about tools, methods and analysis of scenarios of traffic scheduling for Time-Sensitive Networks.

A special consideration will be given to works based on the usage of SMT solvers, as not only they are source for direct comparison with TSNSCHED, but a few of them were of great contribution to the conception of the tool.

3.1 Traffic Control and Packet Scheduling Methods for Time-Sensitive Networks

Given that the TAS is not the only kind of shaper available for TSN networks, it is important to analyze the related work to undersand why it is important to use it, as well as to understand the impact of slot scheduling for maximum efficiency.

The work of Tangamuthu et al. [31] on the analysis of Ethernet-Switch traffic shapers for in-vehicle networking applications, mentioned on the two previous chapters, covers three shapers styles capable of handling traffic of TSN networks: Burst Limiting Shaper, Time-

Aware Shaper and Peristaltic Shaper.

The Burst Limiting Shaper introduces a concept of dynamic priority, where a credit system is created in order to balance the priority of Control-Data Traffic (CDT) class packets and A/B class traffic (which, by default, has a lower priority than packets from the CDT class).

The Time-Aware Shaper, as presented in Chapter 2, has a static schedule for the traffic. One important detail of the schedule used in the paper is that the transmission windows used by the CDT packets is simultaneous on all switches, creating a delay at every hop. Aside from this, the basis for decision of the time slots seems to be superficial, as only simple observations about the constraints of the schedule, such as the requirement of a interval between CDT class gate closing and opening times, are presented.

The Peristaltic Shaper divides the transmission timeline in odd and even phases, where frames are tagged according to the phase in which they are received. The residence time of packets of this shaper depends on the size of the peristaltic phase, determined according to the end-to-end latency of a CDT stream and maximum size of the CDT frame.

These methods, as presented, do not require much effort or preprocessing time to be implemented, but as mentioned in Thangamuthu's work, the results achieved with these shapers in their simplest forms (*i.e.* without further configuration based on the scenario) fail to meet the requirements for 100Mbps Ethernet.

Aiming to deliver a low delay guarantee, low complexity solution, Johannes Specht and Soheil Samii [27] proposed an asynchronous traffic scheduling algorithm for Time-Sensitive Switched Ethernet networks. The algorithm associates flows to queues based on properties such as the previous server visited, the source server and a priority level established by the source, using a new proposed traffic class called Urgency-Based Scheduler (UBS). The evaluation of this work is based only on the end-to-end delay of the packets. Their work is currently under standardization (P802.1Qcr).

As the scheduling approaches for TASs proved to be fruitful for time-critical applications, its research field started to grow. The available methods for synthesis of schedules that are not based on the usage of applied logics is abundant. And since the development of new solutions for scheduling problems have decades of history and solidified works, some of the approaches could be adapted in an attempt to model the instructions of the GCLs.

In his master dissertation, Chowdhury discusses some of the packet scheduling algorithms available to be used on Time-Sensitive Software-Defined Networks (TSSDN) [26]. The approaches for scheduling unicast flows presented in his thesis are divided in two groups: Integer Linear Programming (ILP) and non-ILP scheduling algorithms. The mechanics used in each of these approaches will vary depending if the flow has a fixed path or can use path sets and if it is transmitted over a single or multiple packets. Multicast flows are also discussed here, adding its own set of constraints to the specification of flows and networks.

While presenting a thorough mathematical modeling of the scheduling problems, discussing algorithms for routing, presenting the complexity of the algorithms and evaluating the solutions, Chowdhury evaluates the quality of the algorithms by analyzing how many of the requested flows could be successfully scheduled. The notion of quality of the algorithms also revolves around how many flows can be scheduled as the topologies vary and the time taken to generate these schedules.

Another example of optimization mechanics applied to TSN is the work of Gavriluţ and Pop exploring the similarity of the traffic control in time-sensitive networks with the flow-shop scheduling problem, proposing a synthesis approach based on a Greedy Randomized Adaptive Search Procedure (GRASP) [16]. Such method is capable of deciding the number of Time-Triggered (TT) queues, mapping TT flows to egress port queues and deriving the GCLs of the network. One thing that can be highlighted from this work is the ability to decide the amount of queues used in the port, as some of the other designs on the literature have a limited number of priority queues (or even only one). The scheduling of AVB traffic is also taken in consideration during the solution generation process, increasing the complexity of the problem. Networks with up to 61 flows, 32 end systems and 18 bridges are tested in the experimental evaluation of the approach.

While Gavriluţ's work aims to evaluate if the flows are schedulable, specially when using TT and AVB flows simultaneously, no mention about the flows' jitter is made. The number of time windows available per queue is also not discussed and the topologies presented deal only with unicast flows.

Adapting the problem to a No-Wait Job Scheduling Problem (NW-JSP), handling a packet as a job, Dürr and Nayak developed another solution [13]. Mathematically modeling the problem and using a Tabu Search (a specific type of local neighborhood search) algorithm to find near optimal schedules with a minimal number of guard bands (a technique titled schedule compression, used to avoid wastage of bandwidth), they show that the method proved effective scheduling up topologies with up to 1500 flows, 100 hosts and 20 switches. However, the information about the flows, size of path and number of packets considered in the scheduling process are not disclosed in this work. Finally, the switches are modeled considering that only one queue per port can be used for scheduled traffic.

Also formulating the problem in a ILP framework, Jonathan Falk, Frank Dürr and Kurt Rothermel proposed a solution for routing and scheduling for networks with extension for TSN [14]. It is important to emphasize that this solution does not use predefined routes as it handles the problem as a joint routing and scheduling problem, where both the route and schedules for the packets' transmission are computed in one step. As the routing is handled as part of the problem, the number of vertices in the network graph plays an important path on the time needed to generate a solution. Again, this work uses as a metric for quality of the schedule the runtime of the solver per number of scheduled flows.

Lastly, we mention the study carried out in [34], which indicates that Boolean Satisfiability Problem solvers, also referred to as Satisfiability (SAT) solvers, are capable of performing better than heuristic based solutions for similar problems. Although they are not the same, SMT and SAT solvers share properties and methods of problem solution.

3.2 Usage of SMT Solvers for Handling Scheduling Problems

As scheduling problems can be specified into mathematical/logical problems, the usage of tools such as theorem solvers has been a topic of discussion in the state of the art when it comes to schedule generation. SMTs can offer the users multiple functionalities to aid the theorem specification, offering the creation of modular, simple and reusable theoretical models.

Wilfried Steiner [29], which would later work in the development of a schedule generator for TASs for TSN, demonstrated how it was possible to synthesize schedules for TT multi-hop networks using the YICES SMT solver. Steiner not only shows how YICES, in a out-of-the-box state, can be used to generate schedules for the tested networks, it also shows how, when integrating YICES as a back-end solver of a customized scheduler of TT traffic, larger samples can be efficiently solved by the proposed tool. His work shows that the schedules generated under 30 minutes could have up to 90% maximum utilization on a communication link on scenarios with 1000 packets to be scheduled. It is also mentioned how YICES can be used to verify schedules for TT networks.

Using Steiner's work as a starting point, Silviu Craciunas and Ramon Serna Oliver tackled a similar problem, exploring the simultaneous co-generation of static network and task schedules for distributed systems following a time-triggered paradigm [6]. They present a model of the scheduler using first-order logical constraints to be solved by SMTs and Mixed Integer Programming (MIP) solvers. An incremental method for faster schedule generation is also developed aiming to deal with scalability problems of the tool, enabling its usage for industrial-sized synthetic configurations.

This concept of adding new flows to generated schedules incrementally, where some of the information from the previously generated solution is kept, to generate a new schedule is also reused by Craciunas et al. for solving scheduling problems on TSN TASs.

Sofiene Beji, Sardouna Hamadou, Abdelouahed Gherbi and John Mullins, working on the topics of design of avionic systems, discuss about the improvement of an iterative integration approach used to control the complexity of building architecture designs of such systems connected through TTEthernet, where scheduling the communication of integrated parts is a part of the problem [1]. While it is easier to attain an answer without re-configuring scheduling parameters of the already integrated parts, the recertification process can be costful. [1] proposes a method that uses the YICES SMT solver to find feasible scheduling parameters reducing the cost of integration.

Demonstrating the applicability of SMT solvers on even larger scheduling problems than the ones discussed before, Francisco Pozo, Wilfried Steiner, Guillermo Rodriguez-Navas and Hans Hansson present a technique for synthesis of schedules for extremely large TT networks [24]. They use a decomposition approach to divide the problem in lesser orders of magnitude, reducing the number of constraints necessary to find a solution but still guaranteeing the scenario constraints.

3.3 Schedule Synthesis for Time-Sensitive Networks Based on Theorem Solving

As TSNSCHED is a solution based on a theorem solver, the related work on the field must be observed with a deeper look and can provide the insights for how TSNSCHED was developed.

Concerned with the need of expertise and the consumption of time to configure TASs, Farzaneh, Kugele and Knoll developed a tool where the network modeling can be simply done through a graphical editor [15]. Data is extracted from the model in order to create a knowledge base and constraints can be formulated with the obtained information. The constraints are given to an SMT solver, automatizing the schedule synthesis.

The definition of the problem's formal model explored by Farzaneh is made using the Prolog language, specifically modeling a scenario of in-vehicle applications following a Publish/Subscribe model. The product of this modelling are the constraints used in the definition of the problem to the SMT solver. No constraints regarding the jitter quality are mentioned.

One highlight of this approach is the capability of backtracking in case of failure to produce a feasible schedule. With the proposed tool, a set of variables associated with the transmission streams is used to identify which constraints of the streams could not be satisfied.

Johannes Specht and Soheil Samii, previously mentioned with their work introducing the UBS class to TSN, decided to extend their work to solve the problem of assigning real-time data flows and priority levels to queues [28], as their previous work was limited to the explanation of the newly created class and a analysis of the worst-case latency. As UBS allows a flow to have individual priorities per hop and handles asynchronous traffic, the synthesis problem becomes particularly challenging, and an SMT approach using a cluster based heuristic for dealing with this challenge is proposed. While there is no in-depth specification of how the basic scheduling constraints are built, as well as no means for jitter control, Specht and Samii raise a discussion about methods of efficiency enhancement for smoothly scaling the approach.

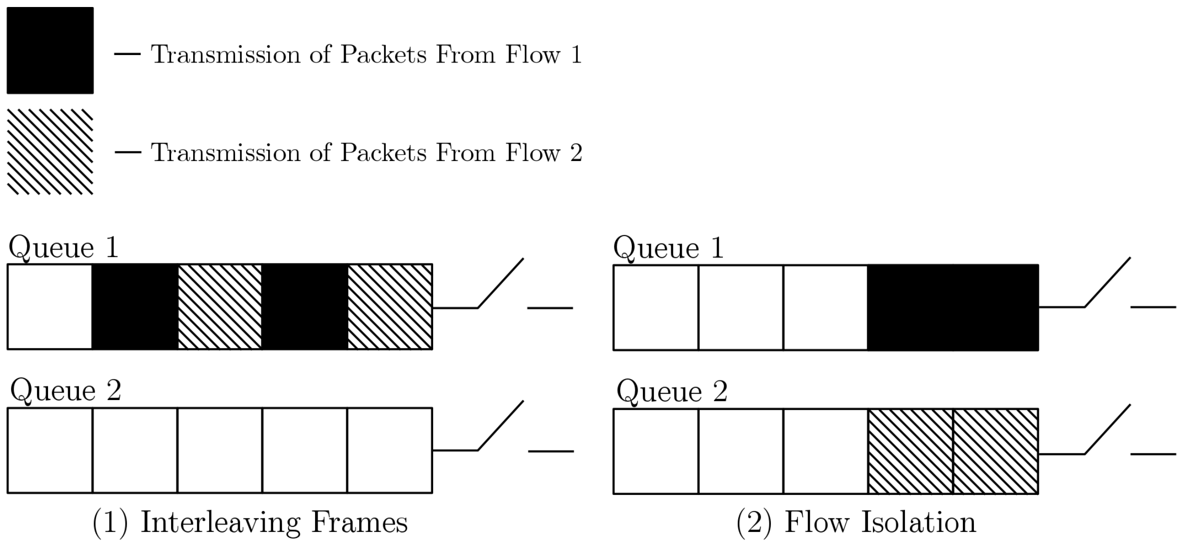
Moving from the research on topics of schedule synthesis for TT networks to TSN TAS schedule synthesis using theorem solvers, Craciunas, Steiner, Oliver and Chmelík, proceeded to publish works describing their approach for building schedules with SMT solvers using Z3 [8][9][7][30], inspiring the work discussed in this thesis. The formal constraints, as well as the evaluation methods, are well defined on the papers that will be discussed next, and an analysis of their work can give an insight for a better understanding of what is to come in the following chapters. From the work available in the state of the art, this is the closest approach to the one used by TSNSCHED.

In [8], a method of schedule synthesis for 802.1Qbv-compliant multi-hop switched networks is presented. The networks are broken down into flows, which, in turn, are composed by physical transmission links of the topology. The links can be shared by different flows and are used to specify the the constraints of a complete flow. The streams described here are periodic, deterministic and have high performance, matching the description of an isochronous flow, presented in [5].

The work not only presents the constraints of the streams are specified, as constraints for the transmission of frames and isolation of frames and flows are also shown. A few of these constraints are used to handle the egress interleaving problem. This problem is characterized by the of loss of determinism due to the placement of frames of two different flows in a queue, such that the interval of transmission of two packets from the same flow in the same queue can be influenced by the presence of frames of another flow in between these two frames. This loss of determinism can happen due to schedules that were generated without taking in consideration latency variation on a flow, ingress policing of the switch and network properties, resulting in streams with high jitter.

The *Flow Isolation* and *Frame Isolation* constraints are used to handle this problem. The *Flow Isolation* constraint specifies that if one flow *A* has frames inside a priority queue, frames of a flow *B* cannot be placed inside that queue until last packet of flow *A* is transmitted. A demonstration can be seen in the Figure 3.1. If the *Flow Isolation* constraint is used, frames cannot be interleaved, so the two flows must be placed in different queues. This constraint limits set of possible values that can be used as a correct answer by the solver and can make the problem harder to solve, as the number of queues per port will now have a much greater influence on the results. It is important to highlight that not every constraint will be a hindrance to the SMT solver, as some assertions may reduce the search space used by the tool, thus accelerating the solution generation procedure. To lessen this problem, the

Figure 3.1: The problem of egress interleaving and the usage of flow isolation.


— Transmission of Packets From Flow 1
— Transmission of Packets From Flow 2
Queue 1
Queue 2
(1) Interleaving Frames
Queue 1
Queue 2
(2) Flow Isolation

Frame Isolation constraint was created, which allows two flows to be placed inside the same queue under the condition that two packets cannot populate the queue at the same time.

Of this work, it is important to emphasize that the main goal was to, with the defined scheduling constraints, produce a correct temporal behaviour such that, once the transmission interval of the packets is obtained, the schedule for the time gates can be easily formulated.

In [7], Craciunas and Oliver present their method in a more concise way, pinpointing the key functional parameters affecting the determinism on TSN networks. Once again, they discuss the formal constraints and present brief comments on the usage of SMT solvers for dealing with such problem. No practical results are shown here.

Following [8], Craciunas, Oliver and Steiner published a paper with an in-depth description of the formal model of a variation of their previously presented method [9]. Here, instead of attempting to schedule each individual frame, windows are scheduled as frames are assigned to them. The method now operates under a configuration where there is one priority queue per port, and what is now queried to Z3 is the opening and closing times of windows of that queue.

Again, Craciunas, Oliver and Steiner attempt to attain a cyclic temporal behaviour for their schedules discovering adequate offsets for sending the first frames of each flow, as well as for the windows opening and closing times.It is not conceived in the formal specification that a frame can arrive before or after the time window where it is sent, making the complexity of the problem greater now that the frame must be immediately forwarded. Also the focus on the formal specification of the constraints allows the reader to obtain a better understanding of what is the problem given to Z3 to solve, but no experimental results are presented.

In [30], Steiner, Craciunas and Oliver discuss about their scheduling technique modeled in [9] once again, but now focusing on the applicability of the tool. After introducing the topics and dissecting the solution generation procedure, they analyze the performance of the scheduling tool and its results. It is important to emphasize that only one priority queue was used in the experiments presented in this paper, and for each priority queue, a set of time windows where the gate of the queue stays open is specified.

This work shares the evaluation criteria presented in [22], where it is possible to measure the quality (latency and jitter) of the schedules generated, as well as the time taken to generate the schedules. For networks with 10 switches and 50 end-systems, topologies with 50 flows and 5 windows per port could take up to 100 minutes to have a schedule generated. The execution time could be enhanced by reducing the number of windows per port utilized in the solution process, but the cost of this can be seen in the average jitter, as, when using the same topology, the average jitter of the schedule generated using 5 windows per queue can be three times lesser than using 1 window per queue.

Still using general purpose SMT solvers, Renan Serna Oliver, Silviu S. Craciunas and Wilfried Steiner discuss how the GCLs' schedules can be expressed via the first-order theory of arrays [22]. In this work, Z3 was again used as the solver for the formulated problem. The results of this research expose the appropriate use of the method, well suitable for small/medium networks, and also exposes the trade-offs faced when using this solving approach. On a network with 40 end-systems, 15 switches, 50 flows handling 211 frames, it can take up to 40 hours with 32 windows per port. As for the size of the network, such scenarios match the sizing of realistic applications such as the ones described in [3].

They also discuss a method of evaluating the quality of the schedules produced by an analysis of the latency and jitter of the use cases, as well as the run time with the varying properties of the desired schedule (such as transmission windows per port).

On an extended abstract, Craciunas, Oliver and Steiner present an online management tool where the methods discussed on their previous works are present as functionalities [10]. The tool allows the user to easily model and configure the network, as well as to generate schedules and deploy the GCLs to the switches.

Chapter 4

TSNSCHED's Formal Model

In this chapter, a novel tool for developing schedules for TSN compliant networks and its formal model are presented. This tool, titled TSNSCHED, is fully configurable by the user, and uses a user-specified number of priority queues per port and techniques for priority modification per hop (VLAN re-tagging [7]) to generate schedules that satisfy given latency and jitter requirements. The tool also supports unicast and multicast flows, which, in other words, are streams of packets that go from one device to another and from one device to many, respectively, following a Publish/Subscribe model, where a device generates and sends messages to one or more listeners, and bandwidth reservation for best-effort traffic.

Another contribution of this work is the formal modeling of the approach, making it easier to comprehend the technique used to solve the scheduling problem without having to understand the low level details of the implementation. The constraints presented in Section 4.4 can also be used to understand the differences between the approach used here from the ones presented in [8] and [9], which are the most similar to our.

4.1 Composition of TSN Networks on the TSNSCHED Perspective

To comprehend how TSNSCHED works, we must take a look at how a TSN network can be modeled as an input for the tool. The network is specified at a high level, where properties of each node and of the data transmission are given as input to build the schedule.

Firstly, a network is composed of nodes of two types: devices and TSN switches. The devices are capable of sending data periodically and have a maximum latency allowed for each packet sent. A device is always connected to at least one switch and can also be used as an end point of a stream.

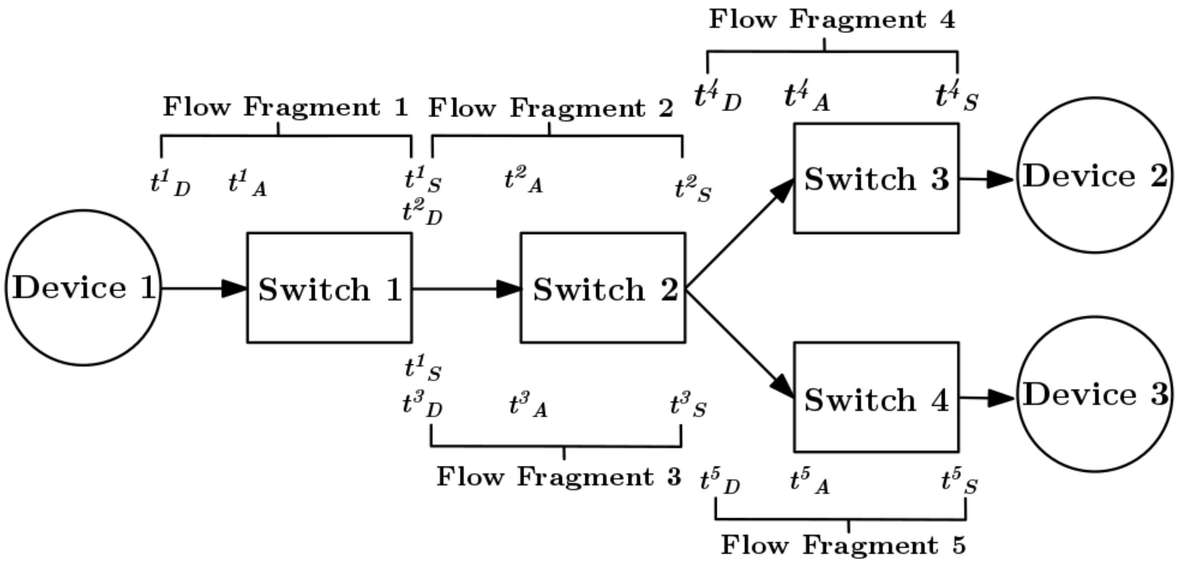
The TSN switches are the nodes capable of relaying data to other switches and build the communication path between two devices. Each switch has a number of ports. Each port possesses a cycle is used to create an one-to-one connection, as TSNSCHED assumes that the middle cannot be shared between devices possessing critical data to be sent. To simulate a full duplex connection, users can create an egress and ingress port on each communication point. Each port has a number of priority queues that can be used for scheduling packets, and for every queue, slots of time will be specified where the gate of that queue will be opened inside the cycle of the switch. No two gates from the same port can be opened at the same time, each gate will open at the same moment(s) in time of every cycle, starting a slot, and each slot has the same duration amount of time every cycle.

A flow is a stream of packets in a TSN network. TSNSCHED is capable of handling unicast and multicast flows following the Publish/Subscribe pattern. Each flow has a starting device sending data periodically to one or more end devices. The path of the stream, composed of switches, is also specified in the flow.

To assert the constraints of the packets in each step of the communication path, a flow is broken into smaller pieces capable of specifying each hop of the stream. The flow fragment is responsible for the representation of all the packets traversing through a switch. The priority tag of the flow in the specific switch that the fragment resides is also possessed by the fragment, and it might change at each hop if the equipment used is 802.1Qci compliant [7], *i.e.* two fragments from the same flow can hold two different priorities. A flow fragment must contain the time where the packet left the last node (Departure Time) in the path, the time where the packet arrived at the current switch (Arrival Time), and the time where the packet left the current switch (Scheduled Time). Notice that a scheduled time of a fragment is the departure time of the next fragment of the flow.

This approach allows multiple patterns to be specified without re-factoring the concepts explained here, since flows can be broken into flow fragments. The difference is that, instead of the methodology suggested in [8], where each Publish/Subscribe multicast flow can be

Figure 4.1: Modeling of a simple TSN network.



split into multiple unicast flows, a tree of flow fragments is created, avoiding fragments to be duplicated and creating overhead for TSNSCHED.

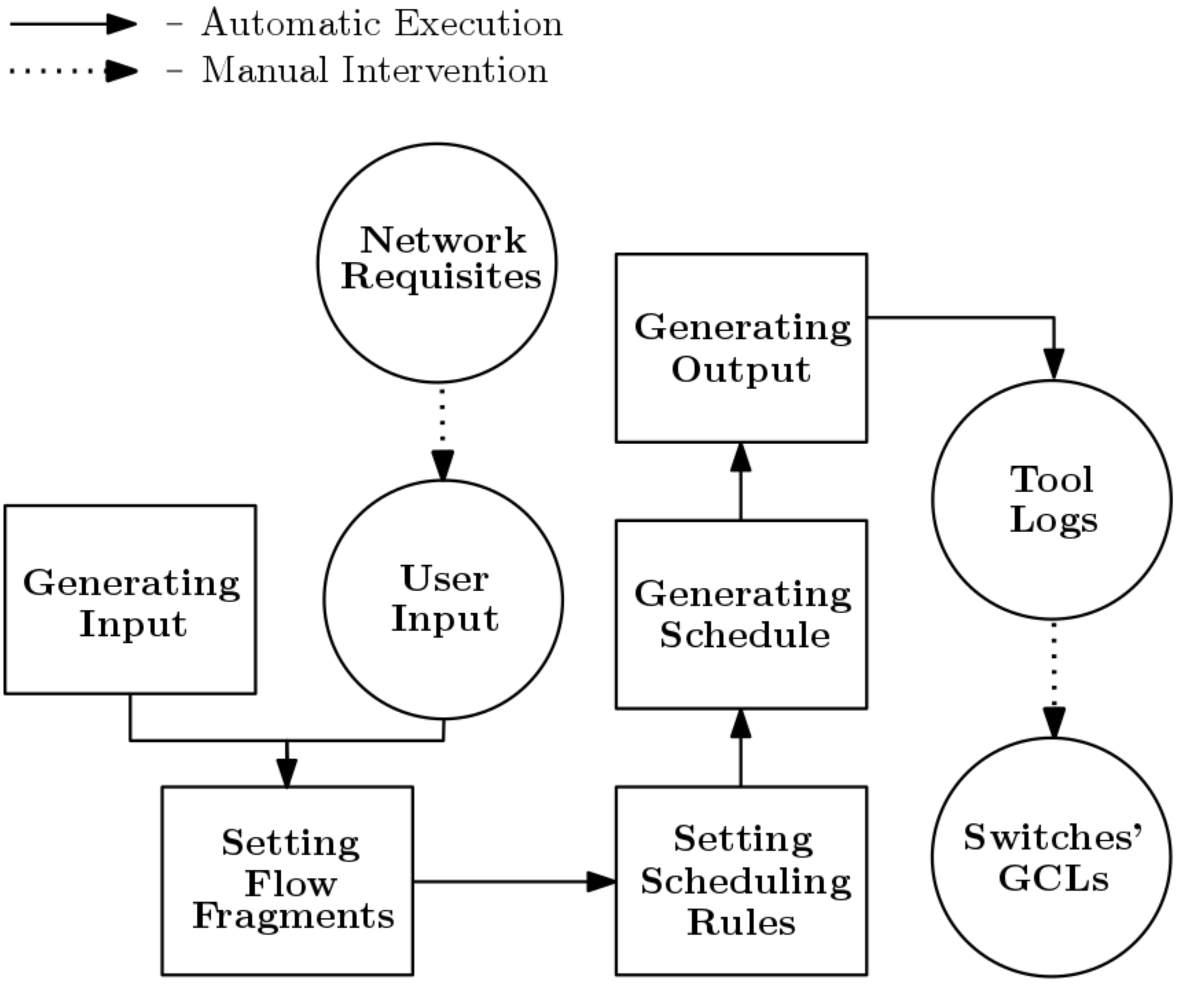
For a better picture of how flows work, a flow can be seen as a chain while each flow fragment is a link in this chain. Now, in a temporal perspective, packets of a flow should arrive at their destination respecting the maximum latency, which can be translated into this analogy as the chain not being able to exceed a certain length. This length is controlled by adjusting the size of every link of the chain, which means that to control the latency of a packet, there must be adjustments in the timing of the hops of the packet in each switch of the path.

In Figure 4.1, a simple abstraction of a network with the topics just discussed is shown. A flow with a starting device named Device 1 sends data in a Publish/Subscribe model to two devices named Device 2 and Device 3. The path is composed by four switches where two of them are shared by the path leading to the two end devices. We can also see on the image that for each port there is a fragment of a flow associated with it. The flow fragments contain the timing information of the packets in the ports of the current switch.

4.2 Scheduling Generation Process

To give a brief overview of how the proposed tool works, TSNSCHED's scheduling generation process can be divided into blocks. As the tasks are executed in a certain order, we can display the process blocks as a diagram to show the tool process overview as shown in Figure 4.2. Bellow, we describe the meaning of each block.

Figure 4.2: TSNSCHED's process overview.



Network Requisites: the process of generating the data used as the schedule for a network starts with the analysis of the network requisites. Following the concepts discussed in Section 4.1, the user can extract the necessary information to conceptualize the network on the perspective of TSNSCHED's input.

User Input: If the user is importing TSNSCHED as a project library, the classes presented on Section 5.1 can be used in order to model a network. In short, the user will specify the scope of the network, providing properties of the switches, devices and flows involved in the communication. Regarding the properties of switches and devices, they include:

- Interval between packets sent by a device;
- Time taken to travel between switches;
- Time taken to process a packet in an egress port (transmission time);
- Maximum cycle duration per switch;
- Minimum cycle duration per switch;
- Maximum priority window transmission time;
- Maximum guard band size per switch;
- Maximum tolerated latency per packet per flow;
- Maximum tolerated jitter per packet per flow.

The tool can also be used as a standalone software, and the class specified by the user modeling the network is given as input.

Generating Input: A scenario generator was created to provide samples of modeled networks for the users, as described later in Section 5.2. It generates topologies according to user-controlled parameters used to define the values mentioned in the list above, and also to define certain properties of the flows. All switches are interconnected as a mesh network. The number of devices is equally divided between the switches and the flows can be unicast and multicast. The number of switches in the path of the flow is also configurable. The main purpose of the topology generator is to create the necessary input to test TSNSCHED's scalability.

Setting Flow Fragments: With the scenario completely modeled as JAVA objects, the first task performed by TSNSCHED is the division of a flow into flow fragments, which includes the conversion of the user-defined data to variables handled by the solver. Due to the structure of the methods provided by its API for assertion creation, the solver used, Z3, cannot handle JAVA primitive type variables, they must be specified in the structures offered by the Z3 JAVA library. Along with this conversion, a number of incognito values are created and bound (if needed) in order to be used as part of the scheduling process. The departure times, arrival times and scheduled times of a flow fragment, which represent the timing variables of a packets, can be used as an example of such values, and will be explored later in Section 4.4.

Setting Scheduling Rules: Once all the Z3 variables are created, the rules that shape the schedule are given to the solver. These rules are formally specified in Section 4.4, but a few of them could not be literally translated to Z3 due to problems regarding the conception of the project, as later explained in Section 4.4.9.

Generating Schedule: With the specified rules, the Z3 solver now attempts to solve the specified scheduling problem. Given the complexity of such problem, most of the execution time of the software is spent here. If the problem cannot be solved using the rules that were previously specified, the user receives a warning informing him of the situation, otherwise TSNSCHED proceeds to the next stage.

Generating Output: After successfully solving the modeled scheduling problem, the tool queries Z3 for the values of the solution found. All the incognito constants created during the conversion of integer and floating point values to Z3 objects now yield a value, once evaluated. These constants include:

- Start of the cycle of each switch;
- Duration of the cycle of each switch;
- Start of time window for transferring priority traffic;
- Duration of time window for transferring priority traffic;
- Priority of the packet in each hop;
- First sending time of a packet of a flow.

The task now is to store these values into the objects used to model the network, so the user can operate them as he prefers.

Tool Logs: A set of logs containing the information of the network is also created. This way, if used as a standalone tool, TSNSCHED still can provide a way to the user to analyze the generated schedule.

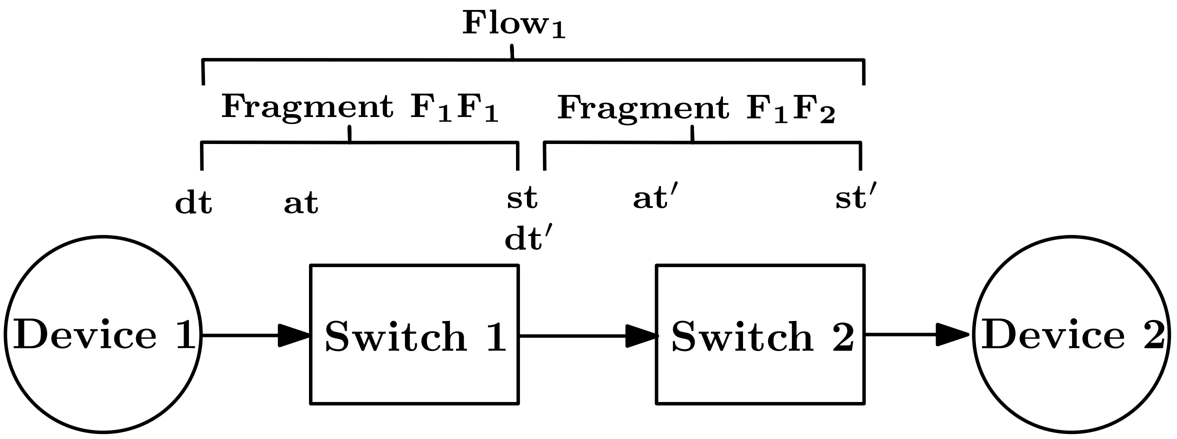
Switches' GCLs: With the information contained in the log, the end-user can give the information given by TSNSCHED as an input to the network configuration, creating the schedules in the switches' GCLs.

As discussed in the future works of this project, the integration of TSNSCHED with other tools might be a way to automate this last step.

4.3 TSNSCHED Application Example

In order to better comprehend how TSNSCHED is built, we can observe a simple example of scheduling of a unicast flow. In this representation, the flow is composed by a starting device, two switches in the communication path and an end device, as pictured in Figure 4.3.

Figure 4.3: A simple unicast flow.



As part of the input, given that in this specific case for simplicity purposes, the two switches share identical configuration, the user specifies the following properties:

- Communicator device periodicity of packet sending: 2000 μs;
- Time taken by the packet to travel between switches: 13 μs;
- Time taken by the packet to be transmitted in each switch: 1 μs;
- Maximum duration of a cycle per switch: 2000 μs;
- Minimum duration of a cycle per switch: 400 μs;
- Maximum size of a time slot: 50 μs;

- Size of the guard band per switch: 20 μs;
- Maximum tolerated latency per packet: 1000 μs;
- Maximum tolerated jitter per packet: 25 μs.

After building the input file to perform the process shown in Section 4.2, the topology is given to TSNSCHED, which, in turn, proceeds to break the flow into two fragments (which can also be seen in Figure 4.3). Once broken into fragments, all the rules of the scheduling problem are set.

Once a model with all the constraints of the problem for the given topology is created, it is given to the solver, which returns the values for the priority of the flow in each switch, the timing variables regarding the cycles, time slots and packet timing variables for the 5 first packets sent. These values are placed in the objects of the input file and also in logs.

The core values of the schedule are:

- Cycle start of Switch 1: 2000 μs;
- Cycle duration of Switch 1: 1991 μs;
- Number of priorities used in the port in Switch 1: 1;
- Priority 1 in Switch 1 time slot start: 0 μs;
- Priority 1 in Switch 1 time slot duration: 50 μs;
- Priority number of Flow 1 in Switch 1: 1;
- Cycle start of Switch 2: 413 μs;
- Cycle duration of Switch 2: 400 μs;
- Number of priorities used in the port in Switch 2: 1;
- Priority 1 in Switch 2 time slot start: 1 μs;
- Priority 1 in Switch 2 time slot duration: 50 μs;
- Priority number of Flow 1 in Switch 2: 1;

- Offset of transmission of Flow 1: 2000 μs.

With the properties shown above, TSNSCHED built a schedule in such a way that every packet belonging to the Flow 1 will be immediately sent after arriving in a switch. In other words, since the path is always the same for every packet of the same flow and packets are forwarded upon arrival, the latency is minimal and the jitter is non-existent.

4.4 Formal Constraints Specification

Before describing the constraints of the scheduler, it must be emphasized that the rules that are about to be shown are a close translation of the rules implemented in TSNSCHED. There are alternatives to the constraints used here, but some are not compatible with the project's design.

Also, TSNSCHED is a flexible tool as it has multiple ways to generate schedules, according to the constraints and functionalities chosen to be used in the scheduling process. These approaches differ mostly in a set of constraints used to restrain the possible values given to the scheduled time of the packets and the definition of the application period, as discussed in Sections 4.4.4, 4.4.5 and 4.4.7.

For generating a schedule, we will be interested in the times when a packet departs, arrives and is scheduled to exit network nodes, as specified in the following definition.

Definition 4.4.1. *Timing Variables:* The timing variables t of a packet[1] are defined by the triple $\langle \mathsf{dt}, \mathsf{at}, \mathsf{st} \rangle$, where:

- dt is the departure time of the packet;
- at is the arrival time of the packet;
- st is the scheduled time of the packet.

Figure 4.3 illustrates the timing variables t_i of two flow fragments, represented in the image by the variables $\langle \mathsf{dt}, \mathsf{at}, \mathsf{st} \rangle$ and $\langle \mathsf{dt}', \mathsf{at}', \mathsf{st}' \rangle$. They intuitively express the times when packets depart, arrive and are scheduled in network nodes. We can chain the events by simply

[1]We abuse slightly of terminology and use packet to also express frames.

adding suitable constraints. For example, in Figure 4.3, the scheduled time of a flow should have the same value of the departure time in the fragment stored in the next hop. This is accomplished by simply adding the constraint: $\mathsf{st} = \mathsf{dt}'$.

Therefore, instead of expressing the properties of the whole flow as a single data-structure, we split flows into Flow Fragments. Recall that the applications assumed here are cyclic and deterministic. This means that in the application period, a bounded number of packets is sent through a flow, and in its corresponding flow fragments. Therefore, flow fragments come with a sequence of time variables, as defined below.

Definition 4.4.2. *Flow Fragment:* A flow fragment ff is a tuple $\langle \mathsf{prt}, \mathsf{T} \rangle$, where:

- prt is the priority of the packets of the flow in the port where the fragment is;
- T is a sequence of time variables $\mathsf{T} = [\mathsf{t}_1, \ldots, \mathsf{t}_n]$. Each timing variables t_i represents the departure, arrival and scheduled time of the i-th packet traversing the flow fragment. We use $\mathsf{T}.size$ to denote the size of T.

The cycle of a port contains the necessary properties to encapsulate all the variables regarding the start, duration and transmission windows on a cycle, as defined below.

Definition 4.4.3. *Cycle:* A cycle c is the 4-tuple $\langle \mathsf{SS}, \mathsf{SD}, \mathsf{s}, \mathsf{d} \rangle$, where:

- SS maps a priority number *i* to a map of real values. This map is a mapping from an index *j* of a slot to a real value, ss, specifying the starting point in the cycle of the slot of index *j* from priority *i*;
- SD maps a priority number *i* to a map of real values. This map is a mapping from an index *j* of a slot to a real value, sd, specifying the duration of the slot of index *j* from priority *i*;
- s is the start of the first cycle;
- d is the duration of a cycle.

For the sake of simplicity, on a cycle with *n* slots per priority, the start of a slot *j* from priority *i* is represented as *SS(i, j)*, and its duration as *SD(i, j)*.

A switch port contains a cycle, the flow-fragments using the port, as well as specifications on the time required to transmit a packet and to travel to node connected to the port.

Definition 4.4.4. *Port:* A port p of a switch is a 5-tuple $\langle \mathsf{c}, \mathsf{PFF}, \mathsf{travelT}, \mathsf{transT}, \mathsf{nc} \rangle$, being the logical representation of a port in the perspective of the scheduling problem.

- c is tuple representing the properties of the cycle of the port;
- PFF is a set of flow fragments that go through that port;
- $\mathsf{travelT}$ is the time taken to travel from this port to its destination and vice-versa;
- transT is the time taken by the port to transfer the packet into the network.
- nc is an integer value indicating how many cycles should be scheduled in that port.

Definition 4.4.5. *Switch:* A switch S is a set of ports.

We denote the set of all switches in the network as SW.

A network is a set of flows F composing all the flows contained in the network. A flow is defined as follows.

Definition 4.4.6. *Flow:* A flow f is as a 4-tuple $\langle \mathsf{FF}, \mathsf{P}, \rho, \phi \rangle$, where:

- FF is a sequence containing the decomposition of a flow into flow fragments ff in the path order. There is one fragment for every switch in the path;
- P is a sequence containing the ordered path of egress ports p of switches from the source to destination node of the flow;
- ρ is the periodicity of a flow, that is, the periodicity in which packets are generated by device source of the flow;
- ϕ is the offset of sending, or the sending time of the first packet of the flow.

With the model defined above, we can establish a set of constraints to shape schedules according to the given rules of the network.

Given the definitions above, for example, $\mathsf{FF}(j).\mathsf{prt}$ is the priority of the fragment j, $\mathsf{FF}(j).\mathsf{T}(k).\mathsf{dt}$ is the departure time of the packet k on the hop of the fragment j, $\mathsf{FF}(j).\mathsf{T}(k).\mathsf{at}$ represents the arrival time of the packet k on the hop of the fragment j and $\mathsf{FF}(j).\mathsf{T}(k).\mathsf{st}$ represents the scheduled time of the packet k on the hop of the fragment j.

As these sets, mappings, sequences and tuples will be heavily used in the next subsections to present the constraints of the scheduling problem, we use the Table 4.1 as a reference table to all network variables used and their meaning. The table also specifies what can be used as input, output or both according to the user's preference in tool's configuration.

There are a few variables which do not explicitly belong to the structure of the network but take part in the scheduling problem, namely the ones that bind restrictions over the results generated by TSNSCHED, and are given as input. The variables and their descriptions can be seen in Table 4.2

Also, for the formal model, the indexing of the sets and sequences start at 1.

4.4.1 Basic Constraints

For a start, we must guarantee that no negative time units are used in the process of generating the schedule. Since all transmission windows are placed within cycles and all packet transmission happen inside these windows, we avoid using negative values for time making the cycles start after a specific point in time, which is 0. We enforce this constraint making sure that every first cycle start of every port ($\mathsf{p.c.s}$) and every offset of every flow ($\mathsf{f}.\phi$) is greater or equal to 0, resulting in the *No Negative Cycle Values* constraint (Equation 4.1) and the *No Negative Frame Time Values* constraint (Equation 4.2).

$$\begin{aligned} &\forall \mathsf{S} \in \mathsf{SW}.\ \forall \mathsf{p} \in \mathsf{S}. \\ &\mathsf{p.c.s} \geq 0 \end{aligned} \tag{4.1}$$

Equation 4.1: *No Negative Cycle Values* constraint.

$$\begin{aligned} &\forall \mathsf{f} \in \mathsf{F}. \\ &\mathsf{f}.\phi \geq 0 \end{aligned} \tag{4.2}$$

Equation 4.2: *No Negative Frame Time Values* constraint.

Also, we define that the sending offset of a flow ($\mathsf{f}.\phi$) is the moment in time where the first packet is sent. In other words, the departure time of the first fragment of a flow

Table 4.1: Formal variables used in the modeling of TSNSCHED

Basic Components of the Network		
SW	Set of switches (S) that compose the topology in a network.	Input
S	Set of ports (p) that compose a switch.	Input
F	Set of flows (f) specifying the communication in the network.	Input
Components of a Port (p)		
c	4-tuple specifying the cycle of a port.	Output
nc	Number of cycles scheduled in that port.	Output
PFF	Set of fragments (ff) that go through a port.	
travelT	Time taken to travel from this port to its destination.	Input
transT	Time taken by the port to process and transmit a packet.	Input
Components of a Cycle (c)		
SS	Mapping of priorities to maps of gate opening times (ss)	Output
SD	Mapping of priorities to maps of transmission windows duration (sd).	Output
s	Start of the first cycle.	In/Out
d	Duration of the cycle.	Output
Components of a Flow (f)		
FF	Sequence of fragments (ff) yield by breaking the flow.	
P	Sequence of ports in the communication path of the flow.	Input
ρ	Periodicity of packet sending of a flow.	Input
ϕ	Offset of the first transmission of a flow (first sending time).	In/Out
Components of a Flow Fragment (ff)		
prt	Priority of the packets that belong to the fragment.	Output
T	Sequence of triplets (t) representing the fragment's timing variables.	Output
np	Number of packets sent (scheduled) on a flow fragment.	In/Out
Components of the Timing Variables (t)		
dt	Packet's time of departure from the last node in the flow's path.	Output
at	Packet's arrival time on current node in the flow's path.	Output
st	Packet's scheduled sending time on current node in the flow's path.	Output

Table 4.2: Formal variables used in the modeling of TSNSCHED

Basic Components of the Network	
$minCycleDuration$	Minimum duration of the cycles.
$maxCycleDuration$	Maximum duration of the cycles.
$gbSize$	Size of the guard band used per port.
$maxSD$	Maximum size of a priority queue transmission window per cycle.
$bestE$	Fraction of a cycle which will be reserved for best-effort traffic (0 to 1).
$maxLatency$	Maximum latency allowed per packet per flow.
$maxJitter$	Maximum jitter allowed per packet per flow.
$numOfPrts$	Number of priority queues available in a port.

($\mathsf{f.FF}(1).\mathsf{T}(1).\mathsf{dt}$), *i.e.* the moment where it leaves the previous node, where, in this case, is the source, is equivalent to the flow's sending offset. Also, periodically, a packet i is sent according to the periodicity of a flow ($\mathsf{f}.\rho$), meaning that after every ρ units of time, another packet will be sent by the source of that flow. This is enforced using the *Packet Departure* constraint (Equation 4.3).

$$\begin{aligned} &\forall \mathsf{f} \in \mathsf{F};\ i \in Z;\ 0 < i \leq \mathsf{f.FF}(1).\mathsf{T}.size. \\ &\mathsf{f.FF}(1).\mathsf{T}(i).\mathsf{dt} = \mathsf{f}.\phi + \mathsf{f}.\rho \times (i-1) \end{aligned} \tag{4.3}$$

Equation 4.3: *Packet Departure* constraint.

This flow offset ($\mathsf{f}.\phi$) also indicates the limit of when a cycle of a port should start ($\mathsf{p.c.s}$). In the following subsections, we will see that the values of the packets' arrival times evaluated by Z3 must fall within a certain range in order to TSNSCHED to work properly. As will be soon presented, a packet must arrive at a switch within the duration of a cycle in order to be scheduled, and, for this reason, every cycle start ($\mathsf{p.c.s}$) must be lesser or equal to the arrival time of the first packet to be processed in this port ($\mathsf{ff.T}(1).\mathsf{at}$). To enforce this, the *Maximum Cycle Start* constraint (Equation 4.4) is used.

To make sure that the solver does not attempt to purposely separate the whole transmission of two different flows with similar periodicity (as in send all the packets of a flow and then send all the packets of another flow), for every flow a maximum offset ($\mathsf{f}.\phi$) must be

$$\begin{aligned}&\forall \mathsf{p} \in \mathsf{f.P}.\ \forall \mathsf{ff} \in \mathsf{p.PFF}.\\&\mathsf{p.c.s} \leq \mathsf{ff.T}(1)\mathsf{.at}\end{aligned} \tag{4.4}$$

Equation 4.4: *Maximum Cycle Start* constraint.

established. This must be done due to requirements in certain systems where the communication of multiple devices must happen in parallel. To do so, it was found suitable to use the periodicity of the flows ($\mathsf{f}.\rho$) as the maximum allowed offset of transmission for each flow, creating the *Maximum Transmission Offset* constraint (Equation 4.5).

$$\begin{aligned}&\forall \mathsf{f} \in \mathsf{F}.\\&\mathsf{f}.\phi \leq \mathsf{f}.\rho\end{aligned} \tag{4.5}$$

Equation 4.5: *Maximum Transmission Offset* constraint.

Without the *Maximum Transmission Offset* constraint, given two flows A and B with the same periodicity, the solver could attempt to schedule all the packets from flow A and then schedule all the packets from flow B after A's packets. For instance, if a cycle of a switch has duration of 200 μs and two flows with the same destination, same periodicity of 200 μs, a maximum allowed latency of 1000 μs and each has three packets to transfer, they could be scheduled in such a way that the solver could try to set the transmission offset of the first flow within the first cycle (the first 200 μs) and the transmission offset of the second flow within the fourth cycle (between 600 and 800 μs). The *Maximum Transmission Offset* constraint prevent this from happening, making the offset of both flows have its value between 0 and 200μs for this example.

This might be problematic if the information transmitted in flows A and B are needed simultaneously in their respective destinations. Furthermore, even if this is not the case, if the scenario is built on the assumption that a complete iteration of a continuous cyclic process can happen after the transmission of *n* packets from both flows, the generated schedule would be unsuitable for the task where, for instance, the transmission of the packets of flow A would have to wait the transmission of all packets from the flow B in order to start its next iteration.

As links in a chain, every flow fragment must be connected to the previous and subsequent flow fragment in the path, if available. Only the departure time of the first flow fragment ($\mathsf{f.FF}(1).\mathsf{T}(k).\mathsf{dt}$) and the scheduled time of the last flow fragment ($\mathsf{f.FF}(\mathsf{f.FF}.size).\mathsf{T}(k).\mathsf{st}$) will not be bound in such way. We assert this connection using the departure and scheduled times of each fragment, making sure that each flow fragment's scheduled time ($\mathsf{f.FF}(j).\mathsf{T}(k).\mathsf{st}$) is equal to the next fragment's departure time ($\mathsf{f.FF}(j+1).\mathsf{T}(k).\mathsf{dt}$) as seen in the *Flow Fragment Link* constraint (Equation 4.6).

$$
\begin{aligned}
&\forall \mathsf{f} \in \mathsf{F}.\ \forall j, k \in Z;\ 0 < j < \mathsf{f.FF}.size;\ 0 < k \leq \mathsf{f.FF}(1).\mathsf{T}.size. \\
&\mathsf{f.FF}(j).\mathsf{T}(k).\mathsf{st} = \mathsf{f.FF}(j+1).\mathsf{T}(k).\mathsf{dt}
\end{aligned}
\tag{4.6}
$$

Equation 4.6: *Flow Fragment Link* constraint.

To better understand this constraint, consider a unicast flow with two switches (and consequently two flow fragments) in its path (such as the one shown in Figure 4.3). The departure times of its first flow fragment will be derived from the transmission offset and the packet periodicity, representing the departure of the packets from the source, and these will be used to generate the scheduled time in the same fragment. The departure times of the second flow fragment, by its turn, represent the times when the packets leave the previous switch in the path, which is equivalent to the first fragment's scheduled times. With concrete values, say that the transmission offset of the flow is equals to 5 μs, which is the value of departure time from the first flow fragment. The packet arrives at the switch at 7 μs and is immediately transmitted, being scheduled to leave at 20 μs. The second flow fragment will have its departure time equals to the scheduled time of the previous fragment, which means that its departure time is 20 μs. Had there been a third switch in the path and the scheduled time of the packet in the second fragment was 35 μs, the departure time of this packet on the third fragment would be 35 μs.

The arrival time of each flow fragment of a flow ($\mathsf{f.FF}(j).\mathsf{T}(k).\mathsf{at}$) is derived from its own departure time ($\mathsf{f.FF}(j).\mathsf{T}(k).\mathsf{dt}$), and depends on the switch it is located. We use this to compute the time to travel between two nodes in the network ($\mathsf{f.P}(j).\mathsf{travelT}$ or $\mathsf{p.travelT}$) inside the timing variables of a flow fragment, taking in consideration variables such as the distance between devices and the travel speed on the communication middle. For this reason, we define the *Arrival Time Value* constraint (Equation 4.7).

$$\begin{aligned}&\forall \mathsf{f} \in \mathsf{F}.\ \forall j, k \in Z;\ 0 < j \leq \mathsf{f.FF}.size;\ 0 < k \leq \mathsf{f.T}.size.\\ &\mathsf{f.FF}(j).\mathsf{T}(k).\mathsf{at} = \mathsf{f.FF}(j).\mathsf{T}(k).\mathsf{dt} + \mathsf{f.P}(j).\mathsf{travelT}\end{aligned} \tag{4.7}$$

Equation 4.7: *Arrival Time Value* constraint.

In other words, if it takes 2 μs to travel between switches *A* and *B*, the packets of the flow fragments in *B* that come from *A* will always have their arrival time equals to their departure time plus 2 μs.

Every fragment of a flow will have a priority ($\mathsf{f.FF}(i).\mathsf{prt}$) that will decide the queue to be occupied by its packets. For the general problem formulation, the number of priority queues per port used is up to 8, which is the default value of the $numOfPrts$ variable, but we can make this a configurable variable per switch with trivial modifications. To ensure that each flow fragment has a priority and it is inside the defined range of queues available, the *Priority Assignment* constraint (Equation 4.8) is used.

$$\begin{aligned}&\forall \mathsf{f} \in \mathsf{F}; i \in Z; 0 < i \leq \mathsf{f.FF}.size.\\ &1 \leq \mathsf{f.FF}(i).\mathsf{prt} \leq numOfPrts\end{aligned} \tag{4.8}$$

Equation 4.8: *Priority Assignment* constraint.

It is important to have in mind that the concept of priority here has no relation to the differentiation of urgency of flows following TSNSCHED's perspective, that is, packets with a higher priority number are not preferred over packet with lower priority numbers. The queues are used to differentiate best-effort traffic from traffic to which transmission time will be reserved, hence the name "priority queue". Having this in mind, the priority of a flow merely indicates the queue in which its packets will be placed, being fit for the solver to decide which one will be used based solely in purpose of identifying the queue.

The number of priorities used in a port also defines how many slots the specific cycle of the port will have. If 8 priorities are used, the mappings SS and SD of each cycle c can have up to 8 elements. There is no way to identify the size of SS and SD with certainty before obtaining an answer provided by the solver, and this happens because the variables that identify which time slots are used are the priority of the fragments, *i.e.* they are part of the output of the problem. By being assigned a priority, the packet can be scheduled to leave at any of the slots specified by the n-th index of SS and SD.

For instance, if one flow fragment in a switch is assigned to a queue with its priority number equals to 3 ($\mathsf{ff.prt} = 3$), the transmission of its packets will happen within one of the time windows identified by the cycle start plus the start and duration of a slot for that priority, but this value is only known after the model is obtained. Simply put, the time slot start and duration in which the packets of ff can be transmitted are identified by the variables $\mathsf{c.SS}(\mathsf{ff.prt}, j)$ and $\mathsf{c.SD}(\mathsf{ff.prt}, j)$, where j is the slot index where the packet is transmitted, and $\mathsf{ff.prt}$ is unknown until the problem is solved.

Another important constraint regarding the priority of the fragments covers the fact that a priority might be fixed for a flow. As mentioned in Chapter 2, in order to change priorities of packets per hop, the switches must be IEEE 802.1Qci compliant, but that may not always be the case. When this operation is not supported, the *Fixed Priority per Flow* constraint (Equation 4.9) is used.

$$\begin{aligned} &\forall \mathsf{f} \in \mathsf{F}.\ \forall \mathsf{ff}_1, \mathsf{ff}_2 \in \mathsf{f.FF}. \\ &\mathsf{ff}_1.\mathsf{prt} = \mathsf{ff}_2.\mathsf{prt} \end{aligned} \tag{4.9}$$

Equation 4.9: *Fixed Priority per Flow* constraint.

Also, as mentioned previously, every port within a switch may share properties regarding the cycle time of start (p.c.s) and duration (p.c.d). While the usage of such configurations is not mandatory, this can be enforced by using the *Equal Cycles Start* and *Equal Cycles Duration* constraints (Equations 4.10, 4.11).

$$\begin{aligned} &\forall \mathsf{S} \in \mathsf{SW}.\ \forall \mathsf{p}_1, \mathsf{p}_2 \in \mathsf{S}. \\ &\mathsf{p}_1.\mathsf{c.s} = \mathsf{p}_2.\mathsf{c.s} \end{aligned} \tag{4.10}$$

Equation 4.10: *Equal Cycles Start* constraint.

The *Equal Cycles Start* constraint is used to allow the simultaneous start of all cycles in a switch, simplifying the configuration of GCLs and improving the readability of TSNSCHED's output.

$$\begin{aligned} &\forall \mathsf{S} \in \mathsf{SW}.\ \forall \mathsf{p}_1, \mathsf{p}_2 \in \mathsf{S}. \\ &\mathsf{p}_1.\mathsf{c.d} = \mathsf{p}_2.\mathsf{c.d} \end{aligned} \tag{4.11}$$

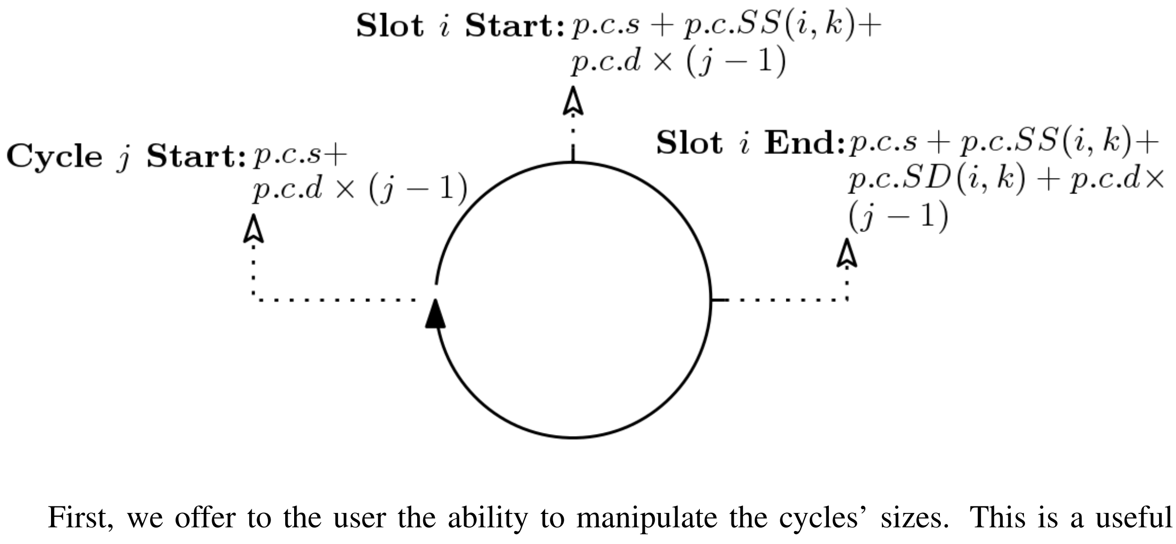
Equation 4.11: *Equal Cycles Duration* constraint.

The *Equal Cycles Duration* constraint is used to allow the creation of simpler schedules to switches, considering possible challenges in the configuration of the GCL of each individual port.

4.4.2 Cycle and Time Slot Constraints

To create the structure of time windows and cycles, we define a set of constraints to specify their behavior to the solver. We present a visual representation of a cycle and one time slot in Figure 4.4, also with the variables used to model them in TSNSCHED's formal model.

Figure 4.4: A representation of cycles and its formal variables.


Slot i **Start:** $p.c.s + p.c.SS(i,k)+$ $p.c.d \times (j-1)$
Cycle j **Start:** $p.c.s+$ $p.c.d \times (j-1)$
Slot i **End:** $p.c.s + p.c.SS(i,k)+$ $p.c.SD(i,k) + p.c.d\times$ $(j-1)$

First, we offer to the user the ability to manipulate the cycles' sizes. This is a useful configuration parameter given that cycles can be adjusted according to capabilities of the switches used. We enforce this by allowing the user to choose upper bound and lower bound values to the duration of the cycles of the ports (p.c.d), namely $minCycleDuration$ and $maxCycleDuration$, respectively. It is interesting to notice that this configuration could be in principle done per switch or per port, according to the usage of the constraints *Equal Cycles Start* and *Equal Cycle Duration* constraints (Equations 4.10 and 4.11). The assertion used is shown in the *Cycle Duration* constraint (Equation 4.12).

$$\begin{aligned}&\forall \mathsf{S} \in \mathsf{SW}.\ \forall \mathsf{p} \in \mathsf{S}.\\&minCycleDuration \leq \mathsf{p.c.d} \leq maxCycleDuration\end{aligned} \tag{4.12}$$

Equation 4.12: *Cycle Duration* constraint.

The start of a time slot ($\mathsf{p.c.SS}(i)$) can only be somewhere between 0 and the end of a cycle ($\mathsf{p.c.d}$) minus the duration of the slot ($\mathsf{p.c.SD}(i)$). Since there can be no guarantee that every slot will be used, the index of the slot will be represented by the priority of the fragments that go through that port, as seen in the *Slot In Cycle* constraint (Equation 4.13).

$$\begin{aligned}&\forall \mathsf{S} \in \mathsf{SW}.\ \forall \mathsf{p} \in \mathsf{S}.\ \forall \mathsf{ff} \in \mathsf{p.PFF}.\ \forall i \in \mathbf{Z};\ 0 < i \leq \mathsf{p.c.SS}(\mathsf{ff.prt}).size.\\&\mathsf{p.c.SS}(\mathsf{ff.prt}, i) \geq 0 \land \mathsf{p.c.SS}(\mathsf{ff.prt}, i) \leq \mathsf{p.c.d} - \mathsf{p.c.SD}(\mathsf{ff.prt}, i)\end{aligned} \tag{4.13}$$

Equation 4.13: *Slot In Cycle* constraint.

Given that a priority of a flow fragment ($\mathsf{ff.prt}$) represents the time slot in which it will be allowed to be transmitted, it is important to specify that if two fragments have the same priority their transmission window, *i.e.* slot start ($\mathsf{p.c.SS}(\mathsf{ff.prt})$) and duration ($\mathsf{p.c.SD}(\mathsf{ff.prt})$), will be the same, as stated by the *Same Priority, Same Slot* constraint (Equation 4.14).

$$\begin{aligned}&\forall \mathsf{S} \in \mathsf{SW}.\ \forall \mathsf{p} \in \mathsf{S}.\ \forall \mathsf{ff}_1, \mathsf{ff}_2 \in \mathsf{p.PFF}.\ \forall i \in \mathbf{Z};\ 0 < i \leq \mathsf{p.c.SS}(\mathsf{ff}_1\mathsf{.prt}).size.\\&\mathsf{ff}_1\mathsf{.prt} = \mathsf{ff}_2\mathsf{.prt} \implies\\&\quad \mathsf{p.c.SS}(\mathsf{ff}_1\mathsf{.prt}, i) = \mathsf{p.c.SS}(\mathsf{ff}_2\mathsf{.prt}, i) \land\\&\qquad \mathsf{p.c.SD}(\mathsf{ff}_1\mathsf{.prt}, i) = \mathsf{p.c.SD}(\mathsf{ff}_2\mathsf{.prt}, i)\end{aligned} \tag{4.14}$$

Equation 4.14: *Same Priority, Same Slot* constraint.

Again, this constraint is built over the fact that slots are referred using the priorities of flows. To the solver, $\mathsf{ff}_1\mathsf{.prt}$ and $\mathsf{ff}_2\mathsf{.prt}$ are concrete values only when a solution to the problem is brought up, which creates the need to establish equivalence between these variables if their values are the same.

Aside from specifying that the same priority leads to the usage of the same space of time, it must be explicit that there can be no overlapping time windows. To do so, first it is specified that a time slot start ($\mathsf{p.c.SS}(i, j)$) cannot be scheduled to be between the start ($\mathsf{p.c.SS}(k, l)$) and end ($\mathsf{p.c.SS}(k, l) + \mathsf{p.c.SD}(k, l)$), as well as a slot end cannot fall within the span of another time slot. We model this asserting that, given two slots from different priorities, one slot can only start after the end or end before the start of another slot. This is enforced with the *No Overlapping Slots* constraint (Equation 4.15).

$$\begin{aligned}
&\forall \mathsf{S} \in \mathsf{SW}.\forall \mathsf{p} \in \mathsf{S}.\forall \mathsf{ff}_1, \mathsf{ff}_2 \in \mathsf{p.PFF}.\\
&\quad \forall i, j \in \mathbf{Z};\ 0 < i \leq \mathsf{p.c.SS}(\mathsf{ff}_1.\mathsf{prt}).size;\ 0 < j \leq \mathsf{p.c.SS}(\mathsf{ff}_2.\mathsf{prt}).size.\\
&\mathsf{ff}_1.\mathsf{prt} \neq \mathsf{ff}_1.\mathsf{prt} \implies\\
&\quad \mathsf{p.c.SS}(\mathsf{ff}_1.\mathsf{prt}, i) \geq \mathsf{p.c.SS}(\mathsf{ff}_2.\mathsf{prt}, j) + \mathsf{p.c.SD}(\mathsf{ff}_2.\mathsf{prt}, j) \vee\\
&\qquad \mathsf{p.c.SS}(\mathsf{ff}_1.\mathsf{prt}, i) + \mathsf{p.c.SD}(\mathsf{ff}_2.\mathsf{prt}, j) \leq \mathsf{p.c.SS}(\mathsf{ff}_2.\mathsf{prt}, j)
\end{aligned} \tag{4.15}$$

Equation 4.15: *No Overlapping Slots* constraint.

A maximum size *maxSD* for a slot is also specified according to the collected network requirements. The duration of the slots ($\mathsf{p.c.SD}(i)$) must be smaller than this specified size according to the *Maximum Slot Duration* constraint (Equation 4.16).

$$\begin{aligned}
&\forall \mathsf{S} \in \mathsf{SW}.\ \forall \mathsf{p} \in \mathsf{S}.\ \forall \mathsf{ff}_1 \in \mathsf{p.PFF}.\ \forall i \in \mathbf{Z};\ 0 < i \leq \mathsf{p.c.SS}(\mathsf{ff}_1.\mathsf{prt}).size.\\
&\mathsf{p.c.SD}(\mathsf{ff}_1.\mathsf{prt}, i) \leq maxSD
\end{aligned} \tag{4.16}$$

Equation 4.16: *Maximum Slot Duration* constraint.

This constraint can be used to aid the optimization of the bandwidth used by priority traffic. If the size of the slots has no upper bound, there is no guarantee that slots will not consume all the time in a cycle, leaving no time to the transmission of best-effort traffic.

Having in mind that switches which cannot use frame preemption must rely on guard bands to minimize the impact caused by non-deterministic behavior on the network (specially coming from best-effort traffic), a constraint to specify the existence of these guard bands is necessary. In short, if two slots are not consecutive, ***i.e.*** the end of a slot ($\mathsf{p.c.SS}(i,j) + \mathsf{p.c.SD}(i,j)$) is not equal to the start of the next slot ($\mathsf{p.c.SS}(k,l)$), then there must be a space of the size of the guard band *gbSize* between them. This is enforced using the *Guard Band* constraint (Equation 4.17).

$$\begin{aligned}
&\forall \mathsf{S} \in \mathsf{SW}.\ \forall \mathsf{p} \in \mathsf{S}.\ \forall \mathsf{ff}_1, \mathsf{ff}_2 \in \mathsf{p.PFF}.\\
&\quad \forall i,j \in \mathbf{Z};\ 0 < i \leq \mathsf{p.c.SS}(\mathsf{ff}_1\mathsf{.prt}).size;\ 0 < j \leq \mathsf{p.c.SS}(\mathsf{ff}_2\mathsf{.prt}).size.\\
&\mathsf{p.c.SS}(\mathsf{ff}_1\mathsf{.prt}, i) \neq \mathsf{p.c.SS}(\mathsf{ff}_2\mathsf{.prt}, j) + \mathsf{p.c.SD}(\mathsf{ff}_2\mathsf{.prt}, j) \wedge\\
&\quad \mathsf{p.c.SS}(\mathsf{ff}_1\mathsf{.prt}, i) > \mathsf{p.c.SS}(\mathsf{ff}_2\mathsf{.prt}, j) \implies\\
&\qquad \mathsf{p.c.SS}(\mathsf{ff}_1\mathsf{.prt}, i) \geq \mathsf{p.c.SS}(\mathsf{ff}_2\mathsf{.prt}, j) + \mathsf{p.c.SD}(\mathsf{ff}_2\mathsf{.prt}, j) + gbSize
\end{aligned} \tag{4.17}$$

Equation 4.17: *Guard Band* constraint.

For instance, if we have a cycle with a time slot A and a time slot B where A comes before B, the end of A must be equal to the start or B or there must be a minimum space of time between them. This space is specified by the size of the guard band. With actual values, if A starts at 20 μs, has 15 μs of duration and the size of the guard band is 10 μs, B must either start at 35μs (at the end of time slot A) or start at a value greater or equal than 45 μs (10 μs after the end of A, the size of the guard band).

In order to manipulate the total size of the time slots used to transfer priority traffic, we need to specify that slots with no flows assigned to them are not used. We express this asserting that their start ($\mathsf{p.c.SS}(i,j)$) and duration ($\mathsf{p.c.SD}(i,j)$) are equals to 0 if no packet from that priority (i) is scheduled to leave during the a specific slot (j), as shown in the *Slot Not Used* constraint (Equation 4.18).

Another constraint regarding the time slots is about the reservation of bandwidth for best-effort traffic. To specify that a certain fraction of the cycle must be reserved for best-effort data transmission, we assert that the sum of the duration of the the slots ($\mathsf{p.c.SD}(i,j)$) cannot be greater than a percentage of the cycle duration (p.c.d). We assert this using a user-configurable variable titled $bestE$, as shown in the *Best-Effort Bandwidth Reservation* constraint (Equation 4.19).

$$\begin{aligned}
&\forall \mathsf{S} \in \mathsf{SW}.\ \forall \mathsf{p} \in \mathsf{S}.\ \forall \mathsf{ff} \in \mathsf{p.FF}.\ \forall i, j, k \in \mathbf{Z};\\
&\quad 0 < i \leq \mathsf{ff.T}.size; 0 \leq j < \mathsf{p.nc};\ 0 < k \leq \mathsf{p.c.SS(ff.prt)}.size.\\
&\quad\quad \neg\exists \mathsf{ff.prt}, i; \mathsf{p.c.s} + (\mathsf{p.c.d} \times j) + \mathsf{p.c.SS(ff.prt}, k) \leq\\
&\quad\quad\quad \mathsf{ff.T}(i)\mathsf{.st} \leq \mathsf{p.c.s} + (\mathsf{p.c.d} \times j) + \mathsf{p.c.SS(ff.prt}, k) + \mathsf{p.c.SD(ff.prt}, k).\\
&\mathsf{p.c.d(ff.prt}, k) = 0
\end{aligned} \tag{4.18}$$

Equation 4.18: *Slot Not Used* constraint.

$$\begin{aligned}
&\forall \mathsf{S} \in \mathsf{SW}.\ \forall \mathsf{p} \in \mathsf{S}.\ 0 < i \leq numOfPrts.\ 0 < j < \mathsf{p.c.SS}(i).size.\\
&bestE \geq 1 - \frac{\sum_{i=1}^{8} \mathsf{p.c.SD}(i, j)}{\mathsf{p.c.d}}
\end{aligned} \tag{4.19}$$

Equation 4.19: *Best-Effort Bandwidth Reservation* constraint.

One last constraint covering the positioning of the time slots regards the order of appearance of them. The *Slot Ordering* constraint (Equation 4.20) guarantees that the slots will be ordered by their index, meaning that a slot of index *j* ($\mathsf{p.c.SS}(i, j)$) can only start after the end of the previous slot, of index *j-1* ($\mathsf{p.c.SS}(i, j-1) + \mathsf{p.c.SD}(i, j-1)$).

$$\begin{aligned}
&\forall \mathsf{S} \in \mathsf{SW}. \forall \mathsf{p} \in \mathsf{S}.\ \forall i, j \in \mathbf{Z};\ 0 < i \leq \mathsf{p.c.SS}.size;\ 0 < j < \mathsf{p.c.SS}(i).size.\\
&\mathsf{p.c.SS}(i, j+1) \geq \mathsf{p.c.SS}(i, j) + \mathsf{p.c.SD}(i, j)
\end{aligned} \tag{4.20}$$

Equation 4.20: *Slot Ordering* constraint.

With this constraint, it is possible to break one of the symmetry problems found during the search for a valid schedule using TSNSCHED. An example of this problem could be found using two different slots of index j and $j + 1$ of the priority i: the SMT solver may end up checking if a valid answer can be built using the start of the slot j equals to a value A and its duration equals to B, similarly, it tests values C and D to the slot start and duration of index $j + 1$. Given that this set of values does not return a valid answer, nothing prevents the solver from assigning C and D to the start and duration of the slot j as well as A and B to the start and duration of the slot $j + 1$. Although the variables have different values, the solutions are the same, implying that the solver may end up testing non-valid schedules multiple times as there is no built-in method to test the equivalence of different answers. For dealing with this, symmetry breaking assertions, such as the *Slot Ordering* constraint, which asserts that the values of the slot start sequences must be ordered according to their indexes, are created.

4.4.3 Core Packet Timing Constraints

The packet timing constraints are the constraints responsible for handling the scheduled time of each packet in the network. They are critical to the definition of how the schedule will be generated as the core of the logic behind the placement of packets in the transmission windows lies here.

Firstly, it must be made clear that a packet cannot be scheduled to leave a switch before the transmission time. To do so, we assert that the scheduled time of a packet ($\mathsf{ff.T}(k)\mathsf{.st}$) is at least greater than the time taken by the port to transmit ($\mathsf{p.transT}$) plus the arrival time ($\mathsf{ff.T}(k)\mathsf{.at}$). This is done using the *Time to Transmit* constraint (Equation 4.21).

$$\begin{aligned} &\forall \mathsf{S} \in \mathsf{SW}.\ \forall \mathsf{p} \in \mathsf{S}.\ \forall \mathsf{ff} \in \mathsf{p.PFF};\ k \in Z;\ 0 < k \leq \mathsf{ff.T}.size. \\ &\mathsf{ff.T}(k)\mathsf{.st} \geq \mathsf{ff.T}(k)\mathsf{.at} + \mathsf{p.transT} \end{aligned} \tag{4.21}$$

Equation 4.21: *Time to Transmit* constraint.

Without this constraint, packets could be scheduled to leave a switch when there is not enough time for transmission in the time slot. For example, if a port transmits a packet of specific size in 10 μs, the *Time to Transmit* constraint is used to assure that the scheduled time of the packet will be greater than the arrival time by at least 10 μs.

It is also important to make sure that the order of transmission of the packet is kept, given that TSN switches cannot transmit two packets that are in the same queue in an order different from the arrival order. To do so, the *FIFO Priority Queue* constraint (Equation 4.22) is used.

This constraint ensures that, for every packet, regardless of which flow it belongs, its transmission will only happen after the ones that arrived before are transmitted. In a tem-

$$\begin{aligned}
&\forall \mathsf{S} \in \mathsf{SW}.\ \forall \mathsf{p} \in \mathsf{S}.\ \forall \mathsf{ff}_1, \mathsf{ff}_2 \in \mathsf{p.PFF};\ k, l \in Z;\\
&\quad 0 < k \leq \mathsf{ff}_1.\mathsf{T}.size; 0 < l \leq \mathsf{ff}_2.\mathsf{T}.size; (\mathsf{ff}_1 \neq \mathsf{ff}_2 \land l \neq k).\\
&\mathsf{ff}_1.\mathsf{T}(k).\mathsf{at} \leq \mathsf{ff}_2.\mathsf{T}(l).\mathsf{at} \land \mathsf{ff}_1.\mathsf{prt} = \mathsf{ff}_2.\mathsf{prt}\\
&\quad \implies \mathsf{ff}_1.\mathsf{T}(k).\mathsf{st} \leq \mathsf{ff}_2.\mathsf{T}(l).\mathsf{st} - \mathsf{p.transT}
\end{aligned} \tag{4.22}$$

Equation 4.22: *FIFO Priority Queue* constraint.

poral perspective, following the terminology used by TSNSCHED, this order can be kept by making sure that one packet's scheduled time ($\mathsf{ff}_1.\mathsf{T}(k).\mathsf{st}$) is lesser than another packet's scheduled time ($\mathsf{ff}_2.\mathsf{T}(l).\mathsf{st}$) only if its arrival time is also smaller ($\mathsf{ff}_1.\mathsf{T}(k).\mathsf{at} \leq \mathsf{ff}_2.\mathsf{T}(l).\mathsf{at}$).

In a more practical way, for flows that must share the same transmission window, there is a rule binding the scheduled time of each packet to every other packet being scheduled. This rule specifies that the first one to arrive is the first one to leave.

To ensure that the transmission of a frame will not happen outside of the boundaries of the time window for its priority queue, it must be explicit in the constraints of the scheduler that the transmission of a frame can only happen between the start and end of a slot. To do so, a packet's scheduled time ($\mathsf{ff.T}(i).\mathsf{st}$) must be equals to any value between the start of a priority slot ($\mathsf{p.c.SS}(\mathsf{ff.prt}, i)$) plus the transmission time up ($\mathsf{p.c.transT}$) until the end of a slot ($\mathsf{p.c.SS}(\mathsf{ff.prt}, i) + \mathsf{p.c.SD}(\mathsf{ff.prt}, i)$), taking in consideration the start of the cycle j ($\mathsf{p.c.s} + \mathsf{p.c.d} \times (j - 1)$) in which it will be transmitted, according to the *Transmit Inside a Time Slot* constraint (Equation 4.23).

$$\begin{aligned}
&\forall \mathsf{S} \in \mathsf{SW}. \forall \mathsf{p} \in \mathsf{S}. \forall \mathsf{ff} \in \mathsf{p.PFF}; i, j, k \in \mathbf{Z};\\
&\quad 0 < i \leq \mathsf{ff.T}.size;\ 0 < j \leq \mathsf{p.nc};\ 0 < k \leq \mathsf{p.c.SS}(\mathsf{ff}).size.\\
&\mathsf{ff.T}(i).\mathsf{st} \geq \mathsf{p.c.s} + \mathsf{p.c.d} \times (j - 1) + \mathsf{p.c.SS}(\mathsf{ff.prt}, k) + \mathsf{p.transT} \land\\
&\quad \mathsf{ff.T}(i).\mathsf{st} \leq \mathsf{p.c.s} + \mathsf{p.c.d} \times (j - 1) + \mathsf{p.c.SS}(\mathsf{ff.prt}, k) + \mathsf{p.c.SD}(\mathsf{ff.prt}, k)
\end{aligned} \tag{4.23}$$

Equation 4.23: *Transmit Inside a Time Slot* constraint.

In other words, if a packet is transmitted in a time window that begins at the 20th microsecond of a cycle and ends at the 30th microsecond of the same cycle, assumed that the transmission time of that packet in that port takes 5 microseconds, the scheduled time of that packet must happen between 25 a 30 microseconds of that cycle.

An optional constraint can be used to prevent two frames from different flows being inside the same priority queue at the same time. This constraint is named *Frame Isolation* Constraint and is proposed by Craciunas, Silviu S., et al. [8]. Its purpose is to reduce the egress interleaving of packets, potentially decreasing the jitter of the flows involved.

To implement the Frame Isolation Constraint in TSNSCHED, it must be enforced that, given two frames *A* and *B* from two different flows with the same priority, one can only arrive after the other left. In other words, the arrival time of A ($\mathsf{ff}_1.\mathsf{T}(k).\mathsf{at}$) must be greater than the scheduled time of B ($\mathsf{ff}_2.\mathsf{T}(l).\mathsf{st}$), or vice versa, as enforced by the *Frame Isolation* constraint (Equation 4.24).

$$\begin{aligned} &\forall \mathsf{S} \in \mathsf{SW}.\ \forall \mathsf{p} \in \mathsf{S}.\ \forall \mathsf{ff}_1, \mathsf{ff}_2 \in \mathsf{p.PFF}; \\ &\quad k, l \in Z; \mathsf{ff}_1 \neq \mathsf{ff}_2; 0 < k \leq \mathsf{ff}_1.\mathsf{T}.size; 0 < l \leq \mathsf{ff}_2.\mathsf{T}.size. \\ &\mathsf{ff}_1.\mathsf{prt} = \mathsf{ff}_2.\mathsf{prt} \implies \mathsf{ff}_1.\mathsf{T}(k).\mathsf{st} \leq \mathsf{ff}_2.\mathsf{T}(l).\mathsf{at} \vee \mathsf{ff}_1.\mathsf{T}(k).\mathsf{at} \geq \mathsf{ff}_2.\mathsf{T}(l).\mathsf{st} \end{aligned} \tag{4.24}$$

Equation 4.24: *Frame Isolation* constraint.

Once the previous constraints regarding the scheduled time of a frame are established, a set of constraints can be used to specify the proper timing for when a frame should leave a switch.

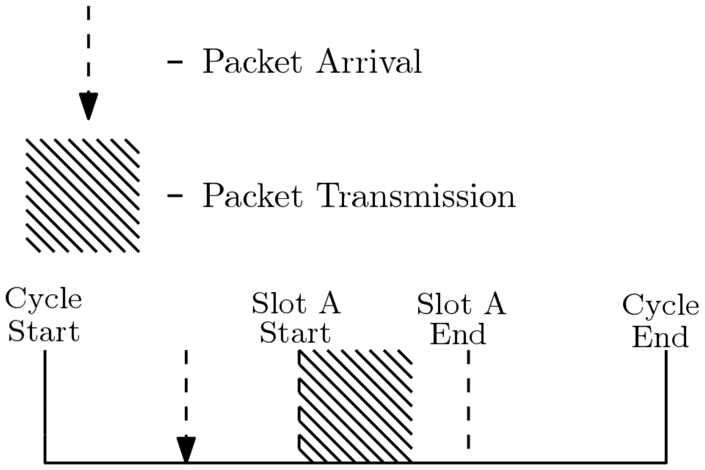
4.4.4 Using Frame Delay

To demonstrate TSNSCHED's flexibility with its constraints, let's assume that we are dealing with a scenario where the time taken to generate a schedule plays an important role in the efficiency of the tool. In this case, we can omit some of the constraints regarding the transmission of the packets in order to deliver an easier problem to the solver, creating the schedule using the frame delay approach. We also assume that the users are willing to add a few extra steps to shape the transmission windows around the packet transmission, making adaptations of the logs generated by TSNSCHED, granted that hours can be saved in this approach.

On the frame delay approach, no other constraint needs to be added to the scheduler. The generated schedule will comply with almost all the rules of operation of the network. The few exceptions are derived from the fact that no constraint specifies that a packet must immediately leave the queue if the gate is open.

Since the constraints used to generate the transmission intervals of packets are rather vague, the problem given to the solver is simpler but this also results in, for instance, transmission of packets starting only moments after the queue's gate opening, which might seem counter intuitive at first.

Figure 4.5: Sending a packet as soon as possible.

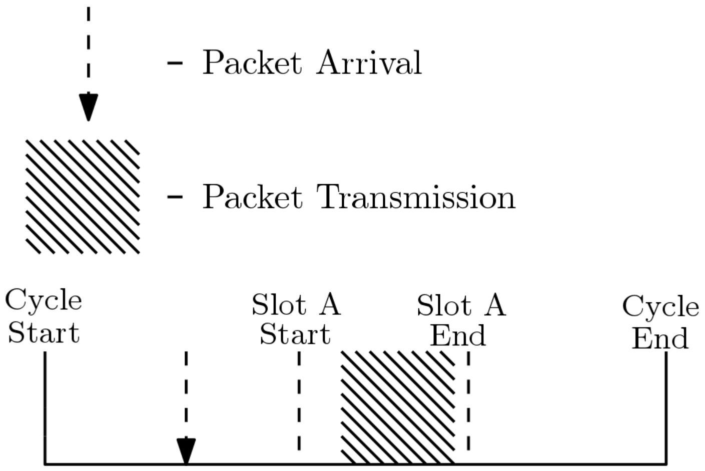


One example of such case would be the scheduling of a flow *A* of priority 1 in a switch *S*. The time window for the transmission of packets belonging to the priority 1 is set to start at 10 microseconds and has 20 microseconds of duration. Assuming that a packet from flow *A* arrives at this switch at the 5th microsecond of the cycle, its transmission would start at the 10th microsecond (when the gate of the queue opens). Given that the transmission of this packet takes 5 microseconds to happen, it would be scheduled to leave at the 15th microsecond of the cycle, as represented in Figure 4.5.

This approach can schedule a packet to leave any moment between the opening and closing of the previously set window, meaning that its transmission can start at, for instance, 20 microseconds after the beginning of the cycle (10 after the start of the slot), as represented in Figure 4.6.

To use this approach, it is assumed that the sending of a packet can be delayed even when the gate of the packet's queue is, in theory, open. There are a number of workarounds which can make this method possible: specifying that time windows must be exactly the size needed to transmit the intended packet in TSNSCHED; obtaining the output of TSNSCHED with time windows of variable sizes and then manually creating time windows around the transmission of specific packets; or introducing a frame sending delay technology on TSN switches.

Figure 4.6: Sending a packet with delay.



4.4.5 Sending Frames As Soon As Possible

In this approach, which is currently used in the standard version of TSNSCHED, the constraints create a more restrictive set of possible answers but are responsible for the generation of schedules where packets are sent without delay, given that the gate of its queue is open. While being an easier method to adapt to real scenarios, there are a few issues regarding the execution time of this approach and the fact that TSNSCHED is only able to use a subset of the actual set of possible valid schedules for the scheduling problem, or in other words, issues with completeness, which will be discussed later.

To ensure that there will be no gaps between the gate opening and the transmission process of a frame, a frame is sent immediately upon arrival on the switch, sent in the the beginning of the next slot if arrived too early or sent right after another frame.

Due to the size of the constraint, it has been broken into 4 predicates *Send After Another Packet*, *If Arrive Before Slot Start*, *If Arrive During Slot* and *No Arrival After Last Slot*, shown respectively in Equation 4.26, 4.27, 4.28, and 4.29, united in the *Send As Soon As*

Possible constraint shown in Equation 4.25.

$$\begin{aligned}
&\forall \mathsf{ff}_1, \mathsf{ff}_2 \in \mathsf{p.PFF}. \\
&SendAfterAnotherPacket \vee \\
&\quad (IfArriveBeforeSlotStart \wedge IfArriveDuringSlot \wedge NoArrivalAfterLastSlot)
\end{aligned} \tag{4.25}$$

Equation 4.25: *Send As Soon As Possible* constraint.

$$\begin{aligned}
&\exists i, j \in \mathbf{Z};\ \mathsf{ff}_1 \neq \mathsf{ff}_2 \wedge i \neq j;\ 0 < i \leq \mathsf{ff}_1.\mathsf{T}.size;\ 0 < j \leq \mathsf{ff}_2.\mathsf{T}.size. \\
&\quad (\mathsf{ff}_1.\mathsf{prt} = \mathsf{ff}_2.\mathsf{prt} \wedge \mathsf{ff}_1.\mathsf{T}(i).\mathsf{at} > \mathsf{ff}_2.\mathsf{T}(j).\mathsf{at})\ \wedge \mathsf{ff}_1.\mathsf{T}(i).\mathsf{st} = \mathsf{ff}_2.\mathsf{T}(j).\mathsf{at} + \mathsf{p.transT}
\end{aligned} \tag{4.26}$$

Equation 4.26: *Send After Another Packet* predicate.

The *Send After Another Packet* predicate (Equation 4.26) means that, for every packet, there must be another with the same priority ($\mathsf{ff}_1.\mathsf{prt} = \mathsf{ff}_2.\mathsf{prt}$) where their transmission happens consecutively. In other words, the scheduled time of every packet ($\mathsf{ff}_1.\mathsf{T}(i).\mathsf{st}$) must be equal to the scheduled time of another packet ($\mathsf{ff}_2.\mathsf{T}(j).\mathsf{st}$) plus the transmission time ($\mathsf{p.transT}$). The constraints regarding the order of transmission will ensure that a packet can only be sent after the previous one is already in the queue. An example of this case is shown in Figure 4.7.

This predicate covers cases where a packet *i* arrives and there is a packet *j* in its priority queue, which is verified by comparing the arrival time of *i* and the scheduled time of *j*. The *FIFO Priority Queue* constraint (Equation 4.22) guarantees that *i* will be properly transmitted in its turn in case there are multiple packets in the queue.

The *If Arrive Before Slot Start* predicate (Equation 4.27) says that if a packet arrives before the beginning of a time slot, it will be transmitted as soon as the slot starts. In other words, if the arrival time ($\mathsf{ff}_1.\mathsf{T}(i).\mathsf{at}$) of a packet falls between the start of a cycle ($\mathsf{p.c.s} + \mathsf{p.c.d} \times (j-1)$) and the start of the first time slot of its priority inside that cycle ($\mathsf{p.c.SS}(\mathsf{ff}_1.\mathsf{prt}, 1) + \mathsf{p.c.s} + \mathsf{p.c.d} \times (j-1)$), its scheduled time ($\mathsf{ff}_1.\mathsf{T}(i).\mathsf{at}$) will be the start

$$\begin{aligned}
&\forall i,j,k \in \mathbf{Z};\ 0 < i \leq \mathsf{ff}_1.\mathsf{T}.size;\ 0 < j \leq \mathsf{p.nc};\ 0 < k \leq \mathsf{p.c.SS(ff)}.size. \\
&(k = 1 \implies ((\mathsf{ff}_1.\mathsf{T}(i).\mathsf{at} < \mathsf{p.c.SS}(\mathsf{ff}_1.\mathsf{prt}, k) + \mathsf{p.c.s}+ \\
&\quad \mathsf{p.c.d} \times (j-1)) \wedge (\mathsf{ff}_1.\mathsf{T}(i).\mathsf{at} \geq \mathsf{p.c.s} + \mathsf{p.c.d} \times (j-1))) \implies \\
&\qquad \mathsf{ff}_1.\mathsf{T}(i).\mathsf{st} = \mathsf{p.c.SS}(\mathsf{ff}_1.\mathsf{prt}, k) + \mathsf{p.c.s}\ + \mathsf{p.c.d} \times (j-1) + \mathsf{p.transT}) \wedge \\
&(k > 1 \implies ((\mathsf{ff}_1.\mathsf{T}(i).\mathsf{at} < \mathsf{p.c.SS}(\mathsf{ff}_1.\mathsf{prt}, k) + \mathsf{p.c.s}+ \\
&\quad \mathsf{p.c.d} \times (j-1)) \wedge (\mathsf{ff}_1.\mathsf{T}(i).\mathsf{at} \geq \mathsf{p.c.SS}(\mathsf{ff}_1.\mathsf{prt}, k-1)+ \\
&\qquad \mathsf{p.c.SS}(\mathsf{ff}_1.\mathsf{prt}, k-1) + \mathsf{p.c.s} + \mathsf{p.c.d} \times (j-1))) \implies \\
&\qquad\quad \mathsf{ff}_1.\mathsf{T}(i).\mathsf{st} = \mathsf{p.c.SS}(\mathsf{ff}_1.\mathsf{prt}, k) + \mathsf{p.c.s}\ + \mathsf{p.c.d} \times (j-1) + \mathsf{p.transT})
\end{aligned} \tag{4.27}$$

Equation 4.27: *If Arrive Before Slot Start* predicate.

$$\begin{aligned}
&\forall i,j,k, \in \mathbf{Z};\ 0 < i \leq \mathsf{ff}_1.\mathsf{T}.size;\ 0 < j \leq \mathsf{p.nc};\ 0 < k \leq \mathsf{p.c.SS(ff)}.size. \\
&((\mathsf{ff}_1.\mathsf{T}(i).\mathsf{at} \leq \mathsf{p.c.SS}(\mathsf{ff}_1.\mathsf{prt}, k) + \mathsf{p.c.SD}(\mathsf{ff}_1.\mathsf{prt}, k)\ + \\
&\quad \mathsf{p.c.s} + \mathsf{p.c.d} \times (j-1) - \mathsf{p.transT}) \wedge (\mathsf{ff}_1.\mathsf{T}(i).\mathsf{at} \geq \\
&\qquad \mathsf{p.c.SS}(\mathsf{ff}_1.\mathsf{prt}, k) + \mathsf{p.c.s} + \mathsf{p.c.d} \times (j-1))) \implies \mathsf{ff}_1.\mathsf{T}(i).\mathsf{st} = \mathsf{ff}_1.\mathsf{T}(i).\mathsf{at} + \mathsf{p.transT}
\end{aligned} \tag{4.28}$$

Equation 4.28: *If Arrive During Slot* predicate.

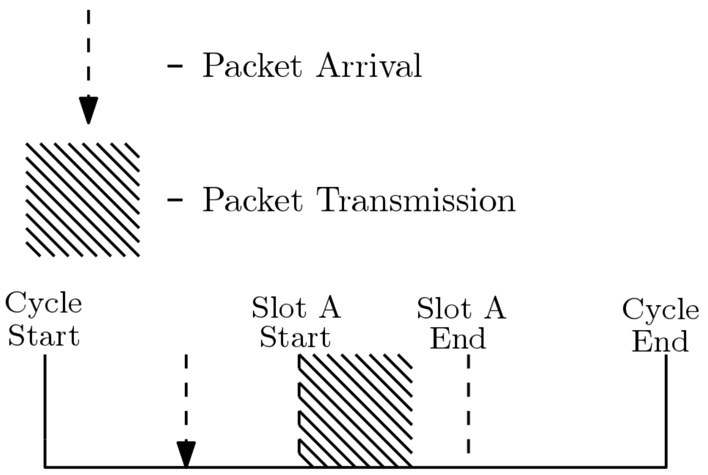
of that slot plus transmission time ($\mathsf{p.c.SS}(\mathsf{ff}_1.\mathsf{prt}, 1) + \mathsf{p.c.s} + \mathsf{p.c.d} \times (j-1) + \mathsf{p.transT}$). An example of this case is shown if Figure 4.5. Also, if configured to use more than one transmission window per priority per cycle, if the packet arrives between two slots, it will be sent on the the beginning of the next one.

The *If Arrive During Slot* predicate (Equation 4.28) says that if a packet arrives during a time slot and there is enough time to transmit, it must be transmitted. In other words, if the arrival time of a packet ($\mathsf{ff}_1.\mathsf{T}(i).\mathsf{at}$) falls between the slot start ($\mathsf{p.c.SS}(\mathsf{ff}_1.\mathsf{prt}, k)$) and slot end ($\mathsf{p.c.SS}(\mathsf{ff}_1.\mathsf{prt}, k) + \mathsf{p.c.SD}(\mathsf{ff}_1.\mathsf{prt}, k)$) of a cycle ($\mathsf{p.c.s} + \mathsf{p.c.d} \times (j-1)$), and there is enough time to transmit within that slot, the scheduled time of that packet ($\mathsf{ff}_1.\mathsf{T}(i).\mathsf{st}$) will be the arrival time plus the transmission time ($\mathsf{ff}_1.\mathsf{T}(i).\mathsf{at} + \mathsf{p.transT}$). An example of this case is shown if Figure 4.8.

$$\begin{aligned}
&\forall i, j \in \mathbf{Z};\ 0 < i \leq \mathsf{ff_1.T}.size;\ 0 < j \leq \mathsf{p.nc}.\\
&(\mathsf{ff_1.T}(i).\mathsf{at} > \mathsf{p.c.s} + \mathsf{p.c.d} \times (j-1)) \wedge (\mathsf{ff_1.T}(i).\mathsf{at} < \mathsf{p.c.s} + \mathsf{p.c.d} \times j)\\
&\quad \implies \mathsf{ff_1.T}(i).\mathsf{st} \leq \mathsf{p.c.SS}(\mathsf{ff_1.prt}, \mathsf{p.c.SS}(\mathsf{ff}).size) +\\
&\quad\quad \mathsf{p.c.SD}(\mathsf{ff_1.prt}, \mathsf{p.c.SS}(\mathsf{ff}).size) + \mathsf{p.c.s} + \mathsf{p.c.d} \times (j-1)
\end{aligned} \tag{4.29}$$

Equation 4.29: *No Arrival After Last Slot* predicate.

Figure 4.7: Transmission of a packet if it arrives while another is in its priority queue.

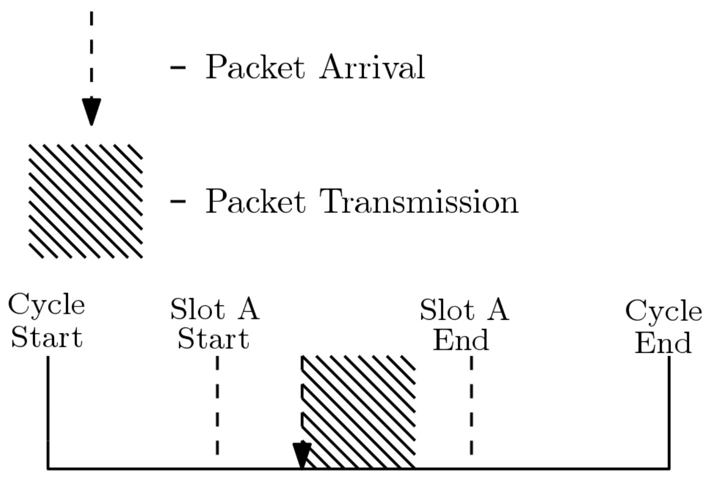


The *No Arrival After Last Slot* predicate (Equation 4.29) says that a packet must not arrive after the last time slot for the priority of its fragment. By allowing packets to arrive after the last time slot, we consider higher values to their lantencies, as well as the possibility of accumulating packets to the following cycles. This is also used in order to limit the solver from trying to obtain invalid answers since the previous predicates do not cover scenarios where packets arrive after the last slot.

The *Maximum Cycle Start* constraint (Equation 4.4) guarantees that one of the implications in predicates *If Arrive Before Slot Start* or *If Arrive During Slot* will have its left side equals to True. For such reason, the solver cannot find a solution where all the implications return false, so either predicate *Send After Another Packet* has to be true, or one of the scenarios covered by the predicates *If Arrive Before Slot Start* or *If Arrive During Slot* must be true while keeping No Arrival After Last Slottrue.

In order to work, these predicates must be united in the constraint shown in Equation 4.25. This means that a packet is sent after another which is in the queue, immediately after

Figure 4.8: Transmission of a packet if it arrives during a time slot.



arriving if there is enough time or at the beginning of a time slot if it arrived too late or too early.

It must be emphasized that even though this method provides trustworthy results (*i.e.*, valid schedules), these constraints do not guarantee the completeness of TSNSCHED. There are cases outside of the boundaries of these constraints where, for instance, a packet cannot be scheduled right after another which is already in the queue because there is no time left in the time slot to transfer. Having this in mind, the solver will try to shape the output of the schedule to model scenarios where such situation is not allowed to happen.

While this has no impact in the reliability of the generated schedule, the process of generating it has a narrower scope of answers.

4.4.6 Jitter and Latency Constraints

To make sure that the schedule fits the needs of the end-system to which it is being developed, there must be guarantees regarding the packet jitter and latency of the flows. These are bound by user specified values defined by *maxLatency* and *maxJitter*, representing the maximum latency and the maximum jitter allowed respectively.

To enforce the allowed latency, a constraint can be created specifying that the difference between the last scheduled time and first departure time must be smaller than *maxLatency*. Since it is considered that the time to reach a device is fixed and defined in the port by the user, there is no need to add the travel time to the scheduled time in this equation. Given that each flow is broken into *n* fragments, this is enforced by the *Maximum Allowed Latency* constraint (Equation 4.30).

$$\begin{aligned} &\forall \mathsf{f} \in \mathsf{F}; i \in Z. \\ &maxLatency \geq \mathsf{f.FF(f.FF.}size\mathsf{).T(}i\mathsf{).st} - \mathsf{f.ff(}1\mathsf{).T(}i\mathsf{).dt} \end{aligned} \tag{4.30}$$

Equation 4.30: *Maximum Allowed Latency* constraint.

Given that the jitter of a flow is the variation of latency, the average latency of that flow must be obtained first, which is calculated using this equation:

$$AvgLatency(\mathsf{f}) = \frac{\sum_i^j \mathsf{f.FF(f.FF.}size\mathsf{).T(}i\mathsf{).st} - \mathsf{f.FF(}1\mathsf{).T(}i\mathsf{).dt}}{j} \tag{4.31}$$

Equation 4.31: Average Latency of Flow F.

With this, the constraint for maximum jitter per packet of a flow can be created. To do so, a constraint must specify that the difference between the latency of a packet and the average latency must not be greater than *maxJitter*, as shown in the *Maximum Allowed Jitter* constraint.

$$\begin{aligned} &\forall \mathsf{f} \in \mathsf{F}; i \in Z : \\ &maxJitter \geq |(\mathsf{f.ff(f.FF.}size\mathsf{).T(}i\mathsf{).st} - \mathsf{f.ff(}1\mathsf{).T(}i\mathsf{).dt}) - AvgLatency(\mathsf{f})| \end{aligned} \tag{4.32}$$

Equation 4.32: *Maximum Allowed Jitter* constraint.

These two constraints can be exemplified using a scenario with a flow *A* that sends three packets. If the maximum allowed latency for this flow is 500 μs and the maximum allowed jitter is 25 μs, a solution where the latencies of the packets are equals to, respectively 495, 500 and 505 microseconds wouldn't be valid, as the latency of the third packet would disrespect the *Maximum Allowed Latency* constraint, even if the jitter has a value of 5. Similarly, if the latencies of the packets are equals to, respectively 400, 450, and 500, even though these values respect the *Maximum Allowed Latency* constraint, the average latency variation of the packets 1 and 3 are equals to 50 microseconds, disrespecting the *Maximum Allowed Jitter* constraint.

4.4.7 Application Period Definition

In order to generate a schedule, it is important to define how many packets must be scheduled within a specific number of cycles. Together with the periodicity of sending and the duration of cycles, we obtain the application period. This element of the scheduling problem can be given as input by the user, but to poorly specify it may imply in the incapability of TSNSCHED to come up with a valid schedule or the generation schedules which, although created following all the constraints of the formal model, fail to behave properly on a real world scenario. An example of such would be if the user whishes to schedule 5 packets of a flow ($\mathsf{ff.np}$) with periodicity of 500μs ($\mathsf{f}.\rho$) in 25 cycles of a specific port ($\mathsf{p.nc}$). If the variables which bind the size of these cycles ($minCycleDuration$ and $maxCycleDuration$) are not configured correctly, it is possible to create schedules that have cycles with, for instance, 510μs of duration. If only one transmission window is configured for this cycle, at every one of its iterations, 10μs of latency may be accumulated for each packet. This delay might not affect any of the 5 scheduled packets, but once installed on a real world scenario planned to run for undefined amounts of time, this accumulated latency will eventually be noticeable. As a consequence, packets may miss their scheduled transmission window, causing the queues to accumulate packets and fail to deliver latency and jitter guarantees.

Similarly, if the user specifies that 10 packets of a flow with periodicity of 1000μs must be scheduled within 3 cycles, and the maximum size of these cycles is, for instance, 500μs, TSNSCHED will fail to generate a solution to the problem, as there are not enough cycles and transmission windows to handle the specified flow.

One trivial way to handle the amount of cycles and packets that must be scheduled is to, if the periodicity of all the flows in the system are the same, set up the cycle size to be equals to the periodicity of the flows. This way, only one packet per flow has to be scheduled in one cycle on a port, and if there is enough time in the transmission windows to transmit all the packets for that specific cycle, considering the latency limitations of the flows, not only the schedule will be valid, the jitter will be non-existent. The problem starts to arise when dealing with flows with different periodicities, when finding the size of the cycle and how many packets per flow must be scheduled becomes a challenge.

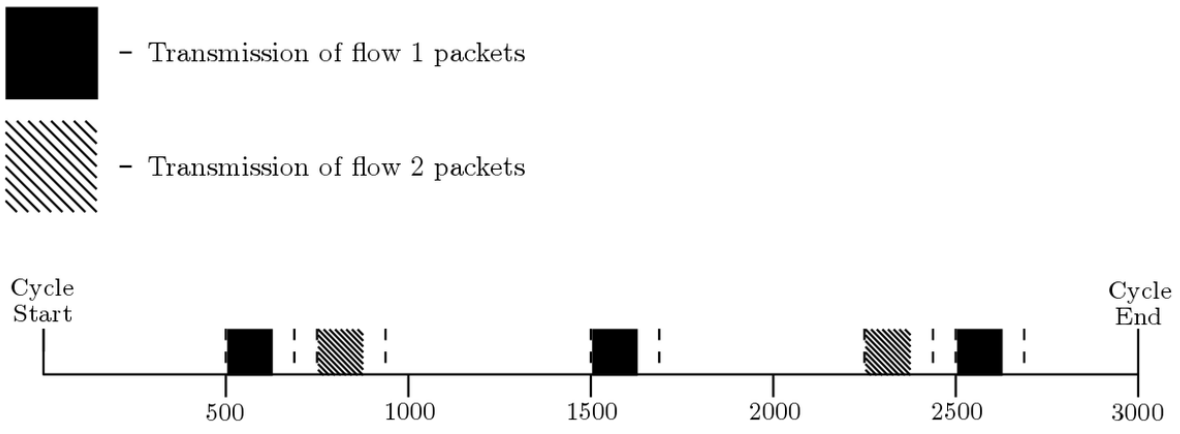
To avoid such problems, we offer methods to automate the calculation of the application periods based solely on the periodicity of the flows.

The Hypercycle Approach

According to [9], an approach for dealing with systems where streams are sourced at multiple rates is to define a cycle fitted according to all the streams on a communication point. The size of this cycle is of at least the *Least Common Multiple* (LCM) of all periods of the flows, and this is defined as the hypercycle.

Explaining it following TSNSCHED's perspective, it is possible to iterate over all the fragments in a port identifying their periodicities. We then calculate the LCM of the periods, which will be the size of the single scheduled cycle. As for the number of packets scheduled per fragment, its value will be the hypercycle size divided by the periodicity of the fragment's flow.

Figure 4.9: Example of a schedule generated using the Hypercycle Approach.


- Transmission of flow 1 packets
- Transmission of flow 2 packets
Cycle Start
Cycle End
500
1000
1500
2000
2500
3000

We see an example of such approach being used on the timeline shown in Figure 4.9, where two flows with different periods are scheduled. Flow 1 has a periodicity of sending (f.ρ) of 1000μs, while flow 2 has a periodicity of sending of 1500μs. The cycle size (p.c.d) is calculated using the LCM of these two values, scheduling a single hypercycle with 3000μs of size. Flow 1 will then have three packets to be scheduled (ff.np) while flow 2 will have two packets scheduled.

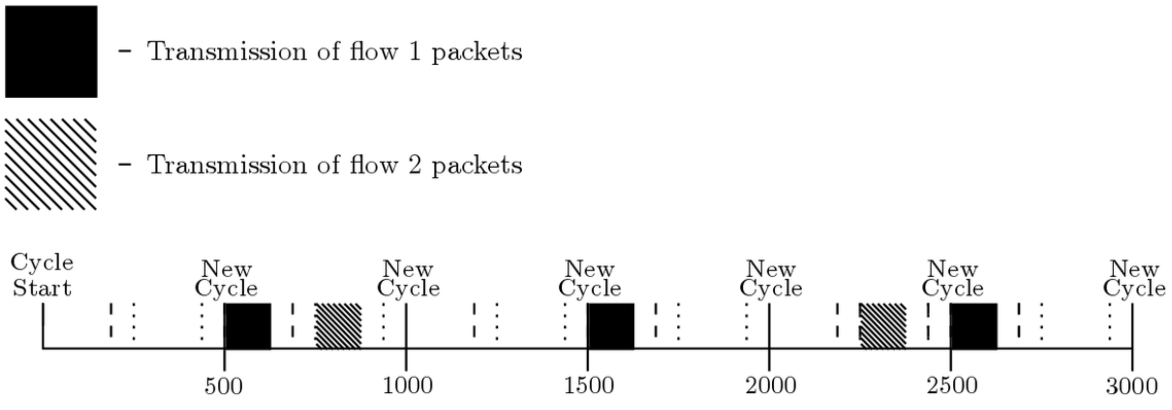
A downside of this method comes from scenarios where its schedules require high amounts of transmission windows on a single cycle. The more slots are used, the more data will be needed to configure the GCLs of the switches, which creates a limiting factor regarding hardware.

The Microcycle Approach

The microcycle method is an extension of the hypercycle approach, taking into consideration the limitations of storage size in the switches GCLs as well as the time taken to generate schedules by TSNSCHED. As further explored in the following chapter, the tool takes longer to generate schedules when more transmission windows per priority per cycle are made available, which implies that one of the ways to achieve better performance using TSNSCHED is to limit the number of transmission windows per priority per cycle to 1. Unfortunately this becomes harder to do in the hypercycle approach due to the need to transmit multiple packets in a cycle, considering the limitations in the size of the transmission windows and the latency and jitter requirements of the flows.

Being aware of these concepts, we take the previously explained approach one step further. After obtaining the size of the hypercycle, we break it into multiple microcycles. The size of a microcycle is equal to the *Greatest Common Divisor* (GCD) of all the flows' periods in a port. The number of cycles scheduled in a port now is the hypercycle size divided by the microcycle size. The number of packets scheduled per fragment per port is calculated in the same way of the hypercycle approach.

Figure 4.10: Example of a schedule generated using the microcycle approach.



Following the same example given for the previously presented approach, we elaborate a schedule using the microcycle method, as shown in Figure 4.10. As the periods of the flows

1 and 2 are of 1000μs and 1500μs respectively, the hypercycle size is 3000μs. Calculating the GCD of the periods, we find that the microcycle size is 500μs. In total, three packets were scheduled for the flow 1 and two packets were scheduled for the flow 2 in 6 cycles of 500μs. Only the information of the first cycle is needed to configure the GCL of a port.

The downside of this method is that, although TSNSCHED is capable of generating a solution faster using the Microcycle approach, some of the cycles will have unused slots. Nevertheless, not only the solution generation process will be faster, the information needed to configure the switches' GCLs is much smaller. It is up to the user to decide if this tradeoff is suitable for the desired application.

4.4.8 Considerations About Multicast Flows

Similarly to the work presented in [8], this model was enhanced to include multicast flows with a trivial extension. Multicast flows could be broken into individual unicast flows where the timing of its flow fragments are bound.

While this extension can be easily done in the formal definition, the implementation of TSNSCHED proved that applying this extension to Z3 was a bit more complex than initially thought.

By definition, the fragment of a flow contains the necessary information regarding the timing of packets and the priority of a flow in one switch. Hence, this low level encapsulation of the flows allows for the simplification of a multicast flow. If the same packet from a flow is sent from a switch to multiple switches, the fragments created in these switches only need to be linked to the previous fragment without transforming multicast flow into multiple bound unicast flows.

Having this in mind, instead of encoding multicasts flow as a whole, we encode modularly each flow fragment and then connect the fragments by simply binding their variables of arrival time and scheduled time. Splitting multicast flows into flow fragments improves the implementation's modularity, readability and modifiability.

4.4.9 Adaptations From The Logical Model

As previously mentioned, the formal definition of the rules used to generate schedules through the solver was presented in a way to translate what was done in the coding of TSNSCHED. Consequently, some of the constraints used were not necessarily explained in the simplest form that they could be, as a literal translation from the formal model to the implementation might not be possible depending on the case.

Initially, according to the *Equal Cycles Start* and *Equal Cycles Duration* constraints (Equations 4.10, 4.11), there is an equivalence between the start and duration of each cycle. This happens when considering the cycle as a part of the switch, not the individual port, while each port has it's own set of rules for the creation of the GCLs data. In the implementation of TSNSCHED, each port has its Cycle object, which, in turn, contains the set of slots start and duration specifically for that port. To make sure that each cycle has the same start and duration, they are made equal with the start and duration given to variables in the switch.

Lastly, there are a few differences in details from the actual implementation of the rules in TSNSCHED from the formal specification previously shown. These have no impact on the meaning of the rules, being mostly related to the order of the predicates and small simplifications in the model. Still, one of these adaptations is quite significant, and was done in the *Send After Another Packet* predicate (Equation 4.26) and the *No Slot Used* constraint (Equation 4.18).

To understand this specific adaptation, note that while JAVA's Z3 API has functions that implement the operations of universal quantification ($\forall$) and existential quantification ($\exists$), the methods in TSNSCHED used to retrieve packet, cycle and slot timing variables cannot be used on them. The *forall* quantifier could be easily replaced with JAVA loops, but the existential quantification operations had to be replaced by sets of operations of logical disjunction. Instead of using the *Send After Another Packet* predicate, the following idea was used: for each packet i, this packet was sent after one packet of the same priority. Say this constraint is being set for the packet 1 in a port where n packets will be transmitted and only one priority queue is available, so packet 1 is sent after packet 2, or packet 3, or packet 4, and so on and so forth until packet n is reached, so long as it is sent after a packet of the same priority. The same is done for packet 2, packet 3, packet 4, going on up to packet n. We use a similar approach to implement the existential quantifier used in the *No Slot Used* constraint

(Equation 4.18).

Chapter 5

Implementation and Experimental Evaluation

In this chapter, we discuss about the implementation, execution, performance analysis and results of TSNSCHED.

We also present a generator of topologies implemented in JAVA code as one of the contributions of this work. The topologies are generated with user-specified number of flows, number of switch and device nodes in the network, number of switches in the path tree and configurable properties of the network and its nodes. The motivation for the creation of this topology generator comes from the absence of benchmarks for the evaluation of scheduling tools for TSN. This will allow for a better understanding of the performance of TSNSCHED.

After covering the JAVA class structure of the tool's implementation, we give an in depth approach of a scenario and a schedule provided by TSNSCHED is analyzed. In this chapter, we also discuss the required tools and technology to run TSNSCHED.

As previously mentioned, the lack of benchmarks for scheduling TSN traffic poses a challenge to the comparison of the effectiveness of TSNSCHED with tools already available. Other tools with similar purpose, while providing useful information for the analysis of the performance and efficiency of the tool, lack details on the topologies, number of scheduled packets, properties of the flows and other variables of the scheduling problem, making it harder to create a direct comparison between TSNSCHED and the work available state of the art. To face this problem, two batches of experiments were created to analyze the tool. The first batch consists of a series of experiments performed with topologies generated by the topology generator were created, aiming to extract information about TSNSCHED's performance in multiple scenarios. These scenarios were also designed with the purpose of comparing various configurations of TSNSCHED, providing valuable insight on the impact of different settings on the performance of the tool.

The second batch of experiments focuses on comparing the two automated application period scheduling approaches, giving a broad comparison of the performance of the two approaches.

The evaluation of the results of these experiments focus on the main constraints of the scheduling problem (jitter and latency) and the execution time of the tool, giving a practical analysis of TSNSCHED's power and performance.

5.1 Tool Implementation

TSNSCHED is implemented in JAVA version 1.8.0_181-b13 using the Z3 API version 4.8.0.0 (over 4.2k lines of code). It takes as input a network topology expressed as switches, devices and flows, as well as the properties of these elements and the performance criteria of the network, specified in a JAVA class with the methods provided by TSNSCHED's library. The criteria used are the maximum allowed latency of packets per flow and the maximum allowed jitter of packets per flow. TSNSCHED reduces the TSN scheduling problem to an SMT problem which is solved using the Z3 SMT solver.

For a modular translation, flows are split into flow fragments, previously described in Chapter 4. If the scheduling problem is shown to be solvable by Z3, then TSNSCHED extracts, from the resulting model, the TSN scheduling configuration for each switch in the network topology. This includes the TSN switch queues' time slots, cycle times, and priority queue associated with each flow.

Currently, TSNSCHED has two modes of operation: standalone and library mode.

As a standalone tool, the user provides a JAVA file containing the specification of a network according to the input required by TSNSCHED and gives it as a parameter to a script. After executing this script, the information about the schedule is stored in logs which can be used by the user to set the remaining properties of the network in the given scenario. As a library, together with the Z3 API, TSNSCHED can be imported on a project to provide its schedule generation capability to other projects. Not only logs are provided, but the information contained in the logs will remain in the objects provided by the user as input.

The code, documentation, importable library file, execution script for the stand alone versions and instructions to use TSNSCHED are available at its github project [12].

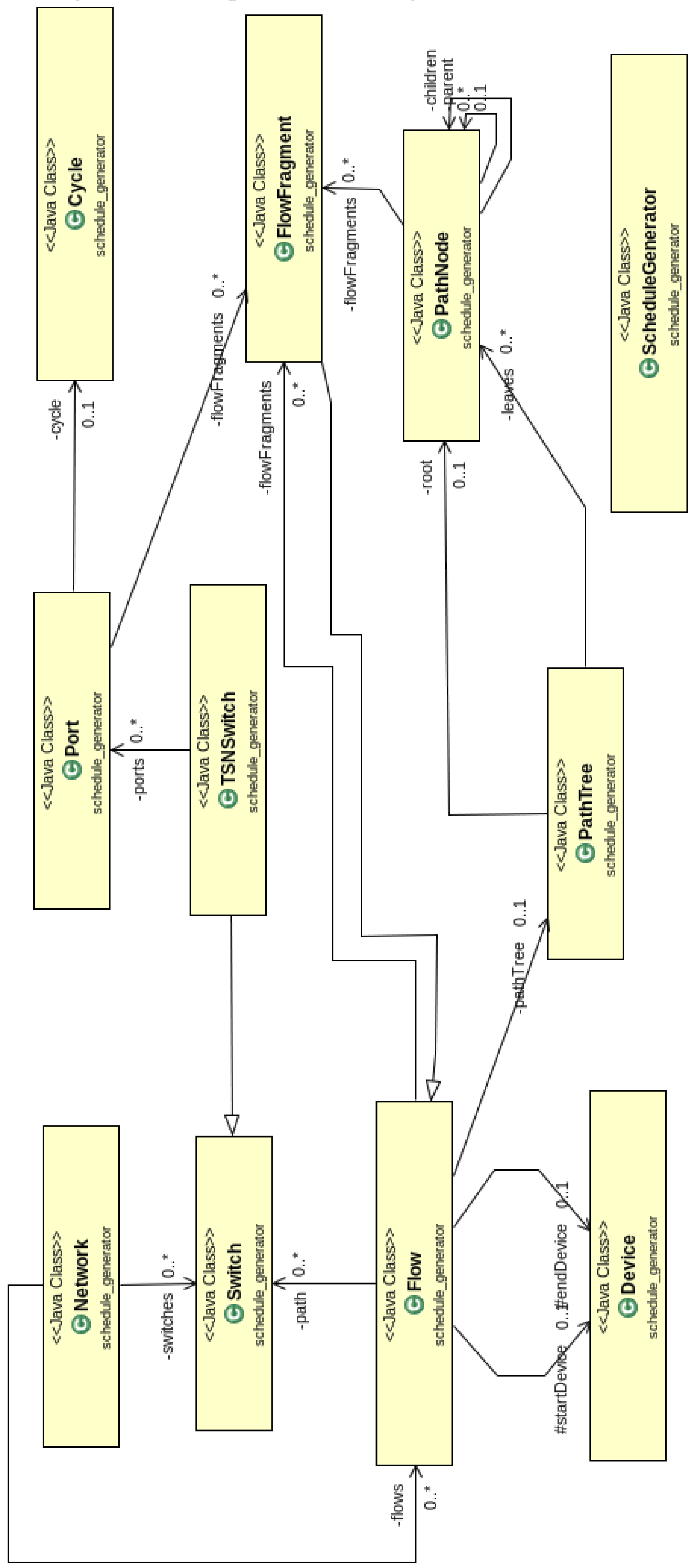
5.1.1 Tool Architecture

The implementation of TSNSCHED and the relationship between its classes can be abstracted using a simplified class diagram presented in Figure 5.1. Since there is an extensive amount of variables and methods in the project, the complete diagram cannot be shown as it does not fit the document.

After modeling the network as an input file, the schedule generation process starts at the ScheduleGenerator class, where the network is given as a parameter to start the scheduling process. The Network class, containing all the data in the topology, will be used to iterate over the flows and switches to create fragments, represented by Flow and TSNSwitch objects, as well as the rules for maximum allowed latency and jitter. The Flows, containing information about the requirements, path, start and end devices of the communication will be broken into fragments, represented by the FlowFragment objects. Also, Flow objects extract many of its properties and requirements from it's start devices, represented by Device objects that contain information such as the periodicity of sending and maximum tolerated latency. The information on the path of a flow is stored in PathTree objects, composed by PathNode objects, which implement basic operations of trees adapted according to its purpose. The FlowFragment objects specifies all the packets that go through a port of a TSNSwitch object, containing the priority of the packets on that port and the timing variables of it's packets. The TSNSwitch, the heir class of Switch, serves as an interface between the flows and ports. Each port objects possess a Cycle, a set of properties regarding the transmission of the packets that go through them, and a set of FlowFragment objects which will be submitted to scheduling rules contained in the port. Cycle objects contain information about their start, duration and a set of timing variables representing the start and duration of time windows for transmission of packets of its priority queues.

Iterations over the flows in the network are performed in order to break them in fragments and store them in the ports of the switches composing the flow's path, followed by the setup

Figure 5.1: Simplified class diagram of TSNSCHED.



of variables used to establish the rules of the schedule. After this process is complete, there will be a new iteration to apply all the scheduling rules to all the fragments in the ports. Finally, a model is requested to the solver, which, if possible, will provide the data for the incognito variables of the scheduling process.

Further information about the individual classes is found in Apendix A.

5.2 Topology generator

Another feature of TSNSCHED's tool-suite is a topology generator. This was needed for evaluating TSNSCHED as there are no openly available flow benchmarks for TSN. Given a desirable number of switches, devices, flows and its respective properties, the topology generator models a network with its flows ready to be used as input for TSNSCHED.

The topologies are pseudo-random, using parameters given by the user to shape flows in different formats while following a desired pattern. By default, the network has 50 end devices and 10 switches. The switches are linked as a mesh network where all the switches are neighbours. The Devices are equally distributed among the switches. No device connects to two switches simultaneously and full duplex communication is simulated by creating two ports communicating two connected nodes in the network.

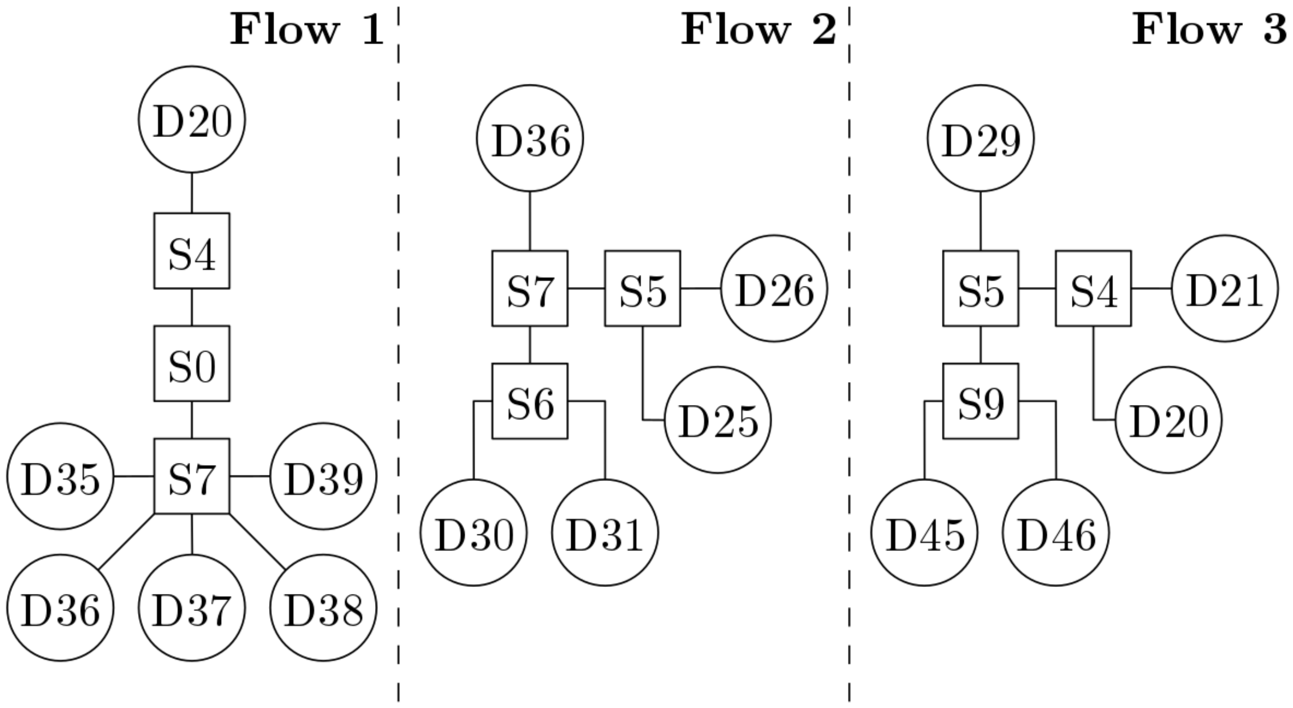
Initially, using a command line, the scenario generator offers the creation of input files for TSNSCHED, taking as parameters the number of flows used, the type of configuration of the flows, the periodicity of each flow and the maximum branching of the path tree, allowing the creation of unicast and multicast flows.

The type of configuration specifies the number of switches composing the path of the flow and the maximum number of subscribers per flow. Three default configurations are available:

- Small Flows, containing 3 switches in the path tree and at most 5 subscribers;
- Medium Flows, containing 5 switches in the path tree and at most 10 subscribers;
- Large Flows, containing 7 switches in the path tree and at most 15 subscribers.

The larger the flow, the harder is the scheduling problem as the depth of the flow and the number of subscribers increase.

Figure 5.2: Example of flows created by the topology generator.



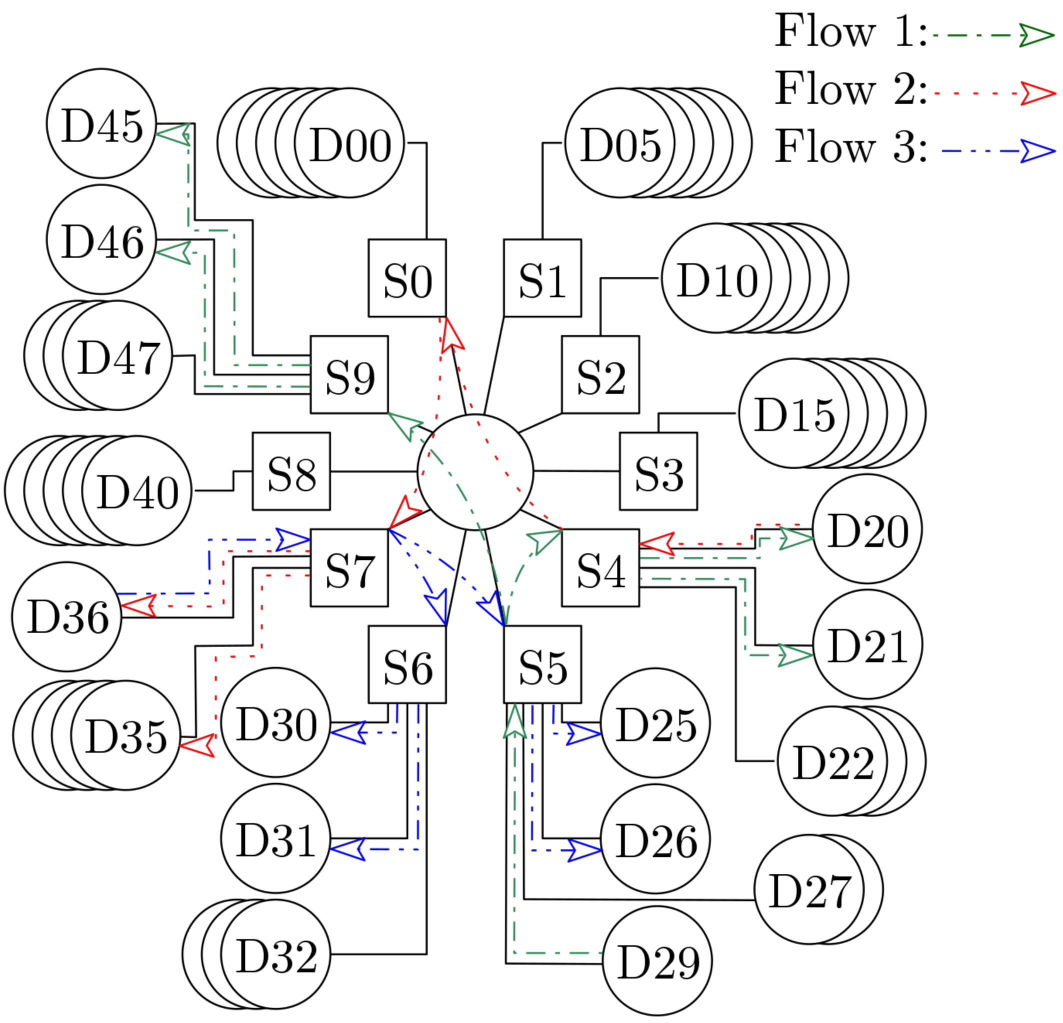
The branching specifies the maximum number of children that a node in the path can have, which influences in the maximum number of subscribers of the flow. For instance, if a user desires to generate a topology with one large flow, it is perfectly possible that, even using a branching parameter greater than one, a unicast flow with 7 switches between the source and the destination is created.

Example of flows in a generated scenario can be seen in Figure 5.2. There are one unicast and two multicast flows in this topology, generated with a maximum branching of 2. All flows are configured as small flows. The first flow has a path with three switches in sequence, while the second and third flows have branches starting on the first switch. The subscribers were equally distributed among the leaves of the tree of switches in the path. The publishers and switches composing the path were picked at random.

In Figure 5.3, these flows are shown on the actual topology. The mesh network has 10 interconnected switches and 50 end-devices equally distributed among the switches.

Topologies similar to this one will be used in part of the process of evaluation of TSNSCHED, which will be seen the following section.

Figure 5.3: Example of topology created by the scenario generator.



5.3 Tool Evaluation

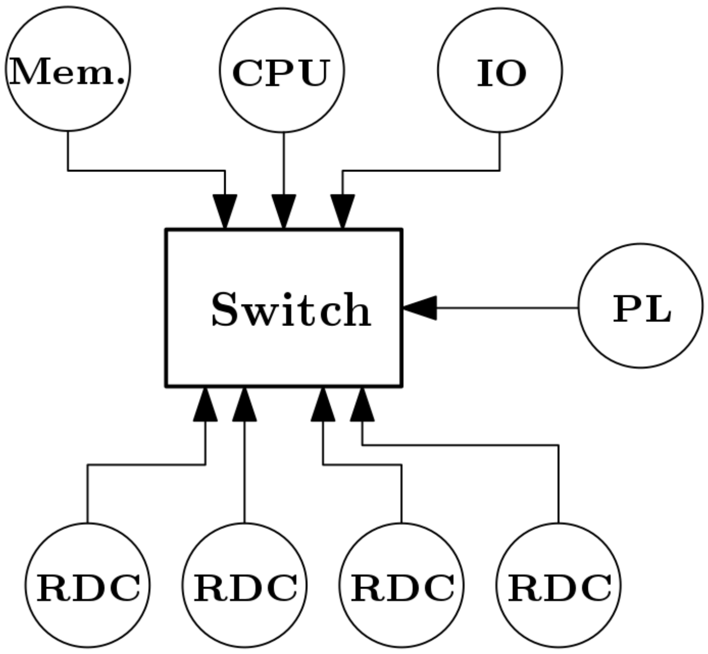
To introduce the practical workings of TSNSCHED, we show a simple case study where information about the individual flows and switches involved can be easily visualized.

Moreover, in order to properly evaluate the tool on experiments of larger scale, we must test its capabilities in different scenarios and understand its boundaries. To do so, we used the scenario generator previously presented to provide topologies for batch testing TSNSCHED.

5.3.1 Case Study

As an introduction to the results obtained with TSNSCHED to real-world applications, we choose to study a simple idealized scenario provided by our partners at Airbus Space, shown in Figure 5.4. As we received a high-level description of the scenario, some of the values and properties of this use case were assumed.

Figure 5.4: Example of topology created by the scenario generator.



In this case, we consider the usage of TSN technology to interconnect components of a system. Using one switch, a set of sensors and actuators, tagged as RDC in the figure, can perform in unison by being coordinated by the CPU. To do so the RDCs must send the data collected by their sensors to the CPU and have a timely response. Failure to comply with the requirements of such communication implies in actuators that should operate together not being synchronized. Having this in mind, the flows created for the communication between the RDCs and CPU have priority traffic.

The CPU will also have to periodically communicate with the memory of the system (tagged in Figure 5.4 as Mem.) as well as to an input/output module (tagged as IO). In order to maintain the quality of service of this application, the traffic between these subsystems is also considered critical.

Lastly, the Mem. must communicate to a module which provides non-critical functionality to the system (tagged as PL). While we consider the traffic between PL and the Mem. as non-critical, the starvation of bandwidth for this traffic may result in problems with the QoS of the system. To avoid such problems, bandwidth must be reserved for best-effort traffic.

Properties of the Communication

The requirements and parameters of the communication flows for the described scenario are as follows:

- Each packet takes 15μs to be transfered;
- Each RDC has a sending periodicity of 20ms;
- The time to travel between RDCs and switches is of 1μs;
- Acceptable latency per packet is of 100μs;
- Acceptable jitter per packet per flow is of 100μs;
- Minimal communication from PL to Mem. of half a cycle; that is, half of the bandwidth available.

Aside from the properties listed above as the specification of the problem, we choose to use a guard band size of 15μs, a lower bound cycle time of 400μs, an upper bound cycle time of 20ms, a maximum size of the transmission windows of 50μs and 5 packets per flow to be scheduled in at most 25 cycles of the switch. All of the devices are connected using one egress and ingress port, hence, the core of this simple scheduling problem is the transmission schedule from the packets from the switch to the CPU, which is the destination of the packets from many of the modules discussed in this problem.

A total of 10 flows were used to model this problem in the input format of TSNSCHED.

Experiment Execution and Results

With the topology and properties specified in the previous subsection, the scheduling problem was solved by TSNSCHED on a Intel(R) Core(TM) i7-4790 CPU @ 3.60GHz machine with 16 GB DDR4 memory. It took 2.3 seconds for Z3 to return a model complying with the constraints of the schedule.

In Figure 5.5, we can see part of the output given by TSNSCHED. The information presented in this figure shows the timing variables of the switch used in this scenario, containing its first cycle start, duration of cycles, fragments contained in each port, priorities used per port, as well as the start and duration of each priority time slot.

In this scenario, we can see that the first port (switch1Port0) has one flow fragment going through it (flow5Fragment1). This port connects the switch to one of the RDCs, and this fragment is part of the flow sending data from the CPU to the RDC. This priority traffic was scheduled to be transmitted inside the priority queue 8, which slot starts at the 9920.20μs of the cycle of the switch, which has a duration of 9999.87μs. The gate of the priority 8 stays open for 15.97μs.

The egress port of the switch which connects to the CPU (switch1Port5) has the most fragments in it, as the flows from all RDCs and Mem. go through. It uses the priorities 3, 5, 6 and 8 to handle these 5 fragments.

The timing of the packets and other information of the fragment, in turn, can be seen in Figure 5.6. The chosen fragment is the flow5Fragment1, being the only one fragment of the flow5, which goes through the switch1Port0 of the switch (the first port in the list of ports shown in Figure 5.5), previously discussed. In the figure, rounding up the values, we can see the offset of the flow (first departure time) was scheduled to be at 20000μs. Since the time to travel is of 1μs, the packet arrives at the switch at 200001μs. Since the cycle of the switch starts at 85.24μs, the gate of the priority 8 opens at 10005.45μs for the first cycle. The second cycle of the switch starts at 10085.12μs, and the gate of the priority 8 opens at 20005.33μs in this cycle.

Figure 5.5: Illustration of TSNSCHED output (Switch information).

```
>>>> INFORMATION OF SWITCH: switch1 <<<<
    Cycle start:     85.242714
    Cycle duration: 9999.879
    Priorities used -
        Port name:        switch1Port0
        Connects to:      dev0
        Fragments:        flow5Fragment1,
        Priority number: 8
        Slot start:       9920.209
        Slot duration:    15.970846
        ------------------------
        [...]
        ------------------------
        Port name:        switch1Port5
        Connects to:      dev5
        Fragments:        flow1Fragment1, flow2Fragment1, flow3Fragment1, flow4Fragment1,
            flow10Fragment1,
        Priority number: 3
        Slot start:       9915.879
        Slot duration:    16.054995
        ------------------------
        Priority number: 8
        Slot start:       0.0
        Slot duration:    23.50163
        ------------------------
        Priority number: 6
        Slot start:       9977.904
        Slot duration:    16.054995
        ------------------------
        Priority number: 5
        Slot start:       9946.934
        Slot duration:    15.970846
```

Figure 5.6: Illustration of TSNSCHED output (Flow information).

```
Flow name: flow5
  Start dev. first t1: 20000
  Start dev. HC: 100
  Start dev. packet periodicity: 20000
  Flow type: Multicast
  List of leaves: dev0,
  Path to dev0: dev5, switch1(flow5Fragment1), dev0,

  Fragment name: flow5Fragment1
      Fragment node: switch1
      Fragment next hop: dev0
      Fragment priority: 8
      Fragment slot start: 9920.209
      Fragment slot duration: 15.970846
      Fragment times-
        (0) Fragment departure time: 20000.0
        (0) Fragment arrival time: 20001.0
        (0) Fragment scheduled time: 20020.33
        ----------------------------
        (1) Fragment departure time: 40000.0
        (1) Fragment arrival time: 40001.0
        (1) Fragment scheduled time: 40020.332
        ----------------------------
        (2) Fragment departure time: 60000.0
        (2) Fragment arrival time: 60001.0
        (2) Fragment scheduled time: 60020.332
        ----------------------------
        (3) Fragment departure time: 80000.0
        (3) Fragment arrival time: 80001.0
        (3) Fragment scheduled time: 80020.33
        ----------------------------
        (4) Fragment departure time: 100000.0
        (4) Fragment arrival time: 100001.0
        (4) Fragment scheduled time: 100020.33
```

Since the first packet of the flow5Fragment1 arrives at 20001μs, it waits until the gate opening of its priority queue to be transmitted. Since the gate of the priority 8 opens at 20005.33μs and it takes 15μs to transmit a packet in the switch, the packet is scheduled to leave at 20020.33μs.

TSNSCHED also returns the average latency and jitter of each flow. For this scenario, the highest average latency was of 99.83μs, and the lowest was of 16μs. The highest jitter per flow was of 0.22μs, and the lowest was of 0. These values comply with the maximum jitter and latency allowed of 100μs.

With these values, it is possible to configure the individual GCLs of the ports in order to use the generated schedule in the implementation of the scenario.

5.3.2 Experiment

To evaluate the capabilities of TSNSCHED, two batches of experiments were performed, providing insightful information about performance, efficiency, flexibility and limitations of the tool. These experiments are described bellow.

Experiment Plan

The idea of the first batch of experiments is to generate a number of test cases with the topology generator and create schedules for them with TSNSCHED. The topologies will have different properties, which will soon be discussed, and are tested using three different approaches: the User-Defined Application Period (UDAP) approach, where the user defines the amount of packets and cycles that will be scheduled; The Hypercycle Approach; and the Microcycle approach, both explained in Section 4.4.7. In the experiments, we use three different variations of the User-Defined Application Period approach: the first one schedules 5 packets to 5 cycles (UDAP[5-5]), the second schedules 5 packets to 10 cycles (UDAP[5-10]), and the third schedules 10 packets to 10 cycles (UDAP[10-10]).

The second batch of experiments aims to compare the Hypercycle and Microcycle approaches individually, testing different topologies with flows of multiple periodicities.

None of the approaches used to perform the experiments are using the *Fixed Priority per Flow*, *Equal Cycles Start*, *Equal Cycles Duration*, *Guard Band*, *Best-Effort Bandwidth*

Table 5.1: Total number of subscribers in the topologies generated for the experiments.

Nº of Flows	1	3	5	10
Small Only	5	15	22	41
Medium Only	10	30	45	100
Large Only	10	40	73	138

Reservation and *Frame Isolation* constraints (Equations 4.9, 4.10, 4.11, 4.17, 4.19 and 4.24, respectively) as not to excessively hinder the execution time of the tool while still providing reliable schedules.

Batch Number 1: Scenarios With Flows of the Same Periodicity

Aiming to simulate industrial scenarios using Publish/Subscribe patterns of communication, we decided to run these tests using multicast flows. We created, using the implemented topology generator, a number of Publish/Subscribe flows on this network, using the branching parameter (maximum number of child switches per switch) equals to 2.

For all the scenarios considered here, we assume a network with 10 TSN switches and 50 physical devices.

We classify the flows in these topologies according to their size (*i.e.* the number of switches and subscribers). We use the same configuration for the size of the flows as previously presented in Section 5.2, using small, medium and large flows.

For each flow, we considered a maximal admissible latency of 1000μs and 25μs jitter, which are adequate for the type of network flow performance described above. Each packet takes 1μs to travel from a node to another and 13μs to be transmitted on a switch. The upper-bound and lower-bound of the size of cycles for the User-Defined Application Period approach are, respectively, 3000μs and 400μs.

In total, 12 topologies were generated. Each topology has a number of flows generated using only one flow size. The same topologies are used for the four variants of the scheduling problem solved by TSNSCHED in the experiments presented here. The number of subscribers according to the size, number and periodicity of scheduled flows used in the experiments is shown in Table 5.3.2.

As for the variants of TSNSCHED, as stated before, this first batch of experiments uses three different variations of the User-Defined Application Period Approach. The first one schedules 5 packets in 5 cycles on each port, the second one schedules 5 packets in 10 cycles and the third schedules 10 packets in 10 cycles. The Hypercycle and Microcycle approaches are also tested in order to provide a comparison of the User-Defined Application Period Approach with an automated application period scheduling method.

A realistic example for a distributed industrial application is the VDMA R+A demonstrator described in [3]. While the exact topology is not provided, this demonstrator contains 26 nodes and 28 unicast flows, including best-effort flows. The size of this application is in the same order as the size of our scenarios. Therefore, TSNSCHED would, in principle, be capable of generating schedules for the VDMA R+A Demonstrator.

5.3.3 Batch Number 2: Scenarios With Flows of Different Periodicities

As the first batch uses scenarios with the same periodicity for all flows, the structure of the solutions provided by the Hypercycle and Microcycle approaches are similar. This happens as the GCD and LCM of all the periodicities of flows are the same, resulting in the same cycle size for both approaches.

For properly comparing the Microcycle and Hypercycle approaches, a different set of experiments was created. The topology shown in Figure 5.7 consists of 5 potential publishers connected to the same switch sending data to the same subscriber. Such arrangement means that the flows will have to compete for bandwidth on the port that connects to the receiver.

With this topology, the idea is to use flows of different periodicities to understand the differences in performance of the Hypercycle and Microcycle approaches. Initially, the periodicities of the flows will be, as we refer to as, closely related, meaning that the hypercycle size (*i.e.*, the lesser common multiple of all flow periods), when divided by the periodicity of the flow, results in a small integer value. Such scenarios cover systems with different integrated parts where the periodicity of all devices involved are not the same, but still they are projected to work in unison; such properties lower the chances of one of the publishers having an arbitrary sending periodicity. For these scenarios, the information about the sending periodicity of publishers, microcycle size (GCD) and hypercycle size (LCM) per use-case is shown in Table 5.2.

Figure 5.7: Topology used to evaluate the Microcycle and Hypercycle approach.

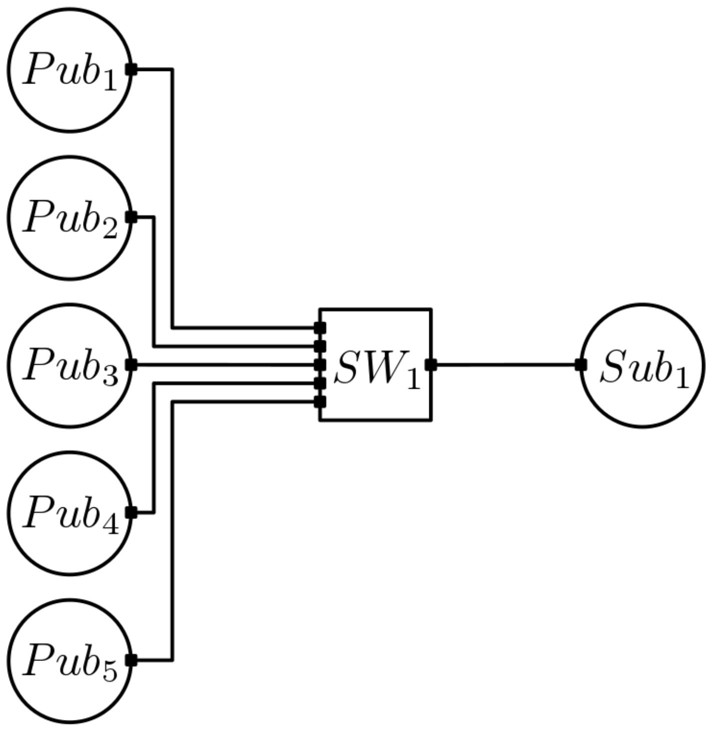


Table 5.2: Sending periodicity of publishers, microcycle size (GCD) and hypercycle size (LCM) per closely related periodicities use-case.

	Pub_1	Pub_2	Pub_3	Pub_4	Pub_5	GCD	LCM
Case A	1000	1000	1000	–	–	1000	1000
Case B	1000	1000	2000	–	–	1000	2000
Case C	1000	1000	1500	–	–	500	3000
Case D	1000	1500	2000	–	–	500	6000
Case E	1000	1500	1500	–	–	500	15000

The second part of this batch of experiments takes in consideration flows with periods that are not closely related, meaning that the approaches will have to schedule more transmission windows and more packets in order to comply with the generated application period. We do not expect to easily find scenarios with such dissonance on the periods of its flows, but these scenarios are useful for understanding the limitations of TSNSCHED. For these scenarios, the information about the sending periodicity of publishers, microcycle size (GCD)

Table 5.3: Sending periodicity of publishers, microcycle size (GCD) and hypercycle size (LCM) per non-closely related periodicities use-case.

	Pub_1	Pub_2	Pub_3	Pub_4	Pub_5	GCD	LCM
Case F	500	800	300	–	–	100	12000
Case G	1880	1400	1350	–	–	10	1776600
Case H	550	800	300	700	–	50	184800
Case I	350	650	750	850	900	50	6961500

Table 5.4: Number of packets and cycles (for the Microcycle approach) scheduled in each scenario of batch 2.

	Number of packets	Number of cycles (Microcycle)
Case A	3	1
Case B	5	2
Case C	8	6
Case D	13	12
Case E	25	30
Case F	79	120
Case G	3530	177660
Case H	1447	3696
Case I	55807	139230

and hypercycle size (LCM) per use-case is shown in Table 5.3.

The total number of packets and cycles (using the Microcycle approach) scheduled in each scenario is shown in Table 5.4. The number of cycles scheduled using the Hypercycle approach is not presented in the table as it is always 1.

For each flow, we considered a maximal admissible latency of 1000μs and 25μs jitter, which are adequate for the type of network flow performance described above. Each packet takes 1μs to travel from a node to another and 13μs to be transmitted on a switch. Specially for the use-case G, we lower the transmission time to 10μs in order to be able to fit the whole transmission of the packet inside a cycle, as TSNSCHED does not support transmitting a single packet in two different cycles.

Experiment execution

The experiments were carried out on 8 virtual cores of a Intel(R) Xeon(R) CPU E5-2620 v3 @ 2.40GHz processor with 16 GB DDR4 memory. We set a timeout of 80 hours for each scenario in our set of experiments. Since schedules need to be generated only once, such long execution times is acceptable in practice.

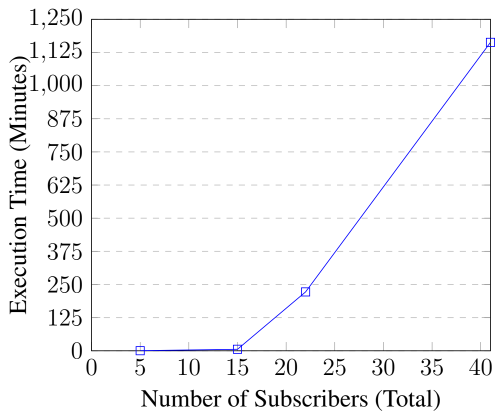
The execution time of the experiments varied also as expected, following an exponential pattern, as depicted by Figures 5.8, 5.9, 5.10, showing the execution times of the approaches according to the number of subscribers in the topology.

As for the differences in the time taken to generate schedules in the used approaches, we stress that the execution time of the tool is heavily influenced by the number of constraints used to specify a problem, as well as the number of packets and cycles that are scheduled in the problem. For instance, given that the UDAP[5-10] approach schedules more packets and cycles than the UDAP[5-5] approach, it is understandable why it would have a higher execution time. The same can be seen comparing the UDAP approaches to the Hypercycle and Microcycle approaches when all the flows have the same sending periodicity. One of the most expressive examples of this can be seen comparing Figure 5.8 (a) and Figures 5.8 (d) and (e). On the scenarios scheduled using the UDAP[5-5] approach, the tool managed to schedule 41 subscriber nodes in close to 1100 minutes, while the Hypercycle and Microcycle approaches both managed to schedule the same scenario in less than a second; an improvement of aproximately 5 orders of magnitude. A reflection of the hindrance that comes with scheduling more packets and more cycles is seen in the fact that only the UDAP[5-5], among all the scheduling methodologies where the periodicity is defined by the user, is capable of scheduling all the 4 small flow scenarios, also timing out on larger use cases.

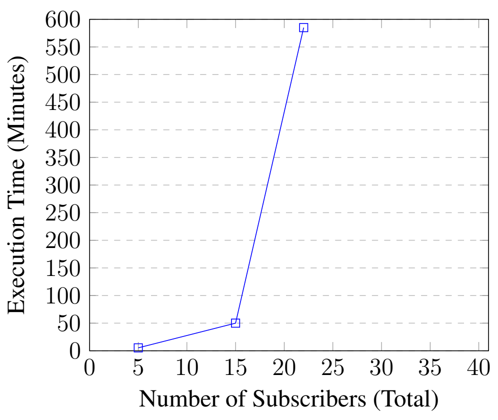
It is also interesting to point out the proximity of the execution times for the Hypercycle and Microcycle approaches, given the similarity of the approaches' structure of solutions for scenarios with a single period for its flows.

Less surprising was the fact that TSNSCHED had more difficulties in generating schedules for scenarios with larger flows. This was expected as the complexity of the scheduling problem increases with the size of flows, *e.g.* number of switches to be configured, showing

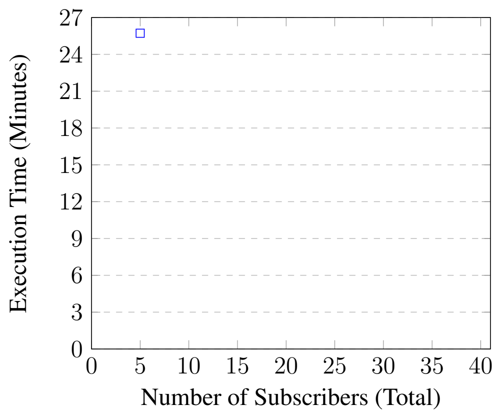
Figure 5.8: Execution time per number of subscribers on scenarios using only small flows.



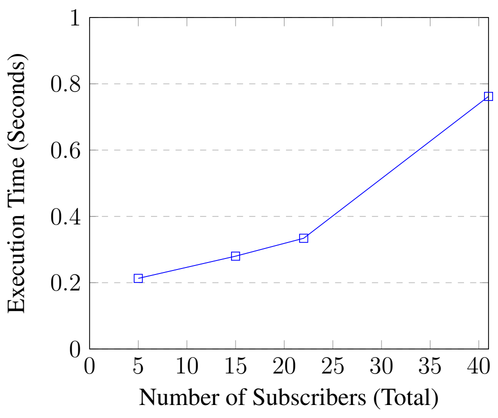
(a) UDAP [5-5].



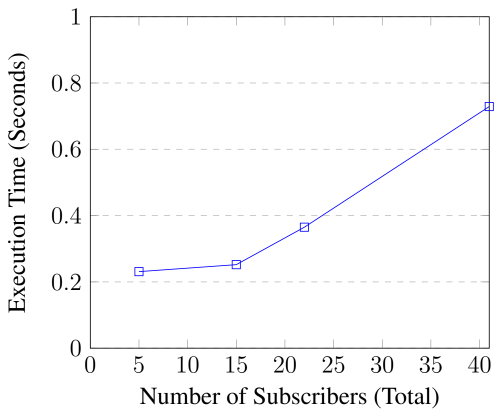
(b) UDAP [5-10].



(c) UDAP [10-10].

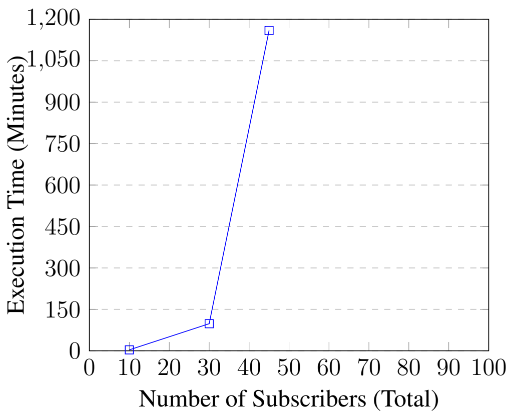


(d) Hypercycle.

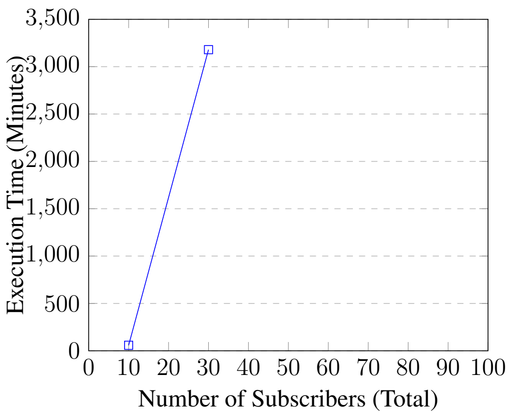


(e) Microcycle.

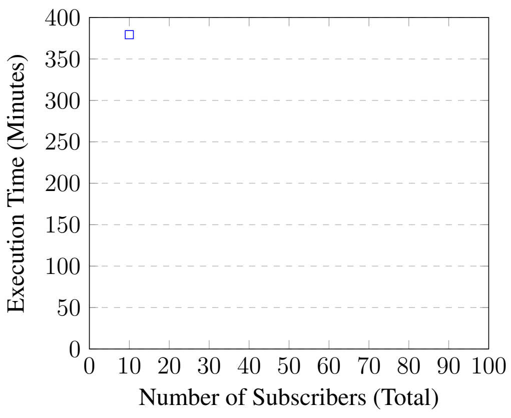
Figure 5.9: Execution time per number of subscribers on scenarios using only medium flows.



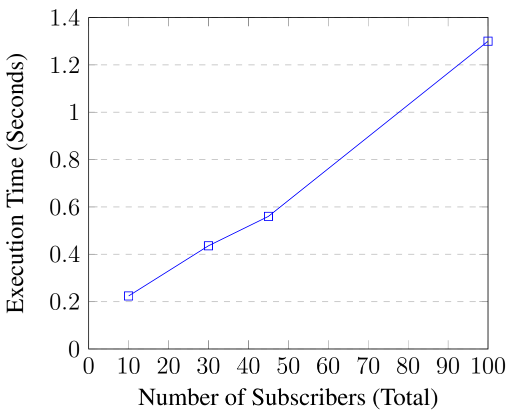
(a) UDAP [5-5].



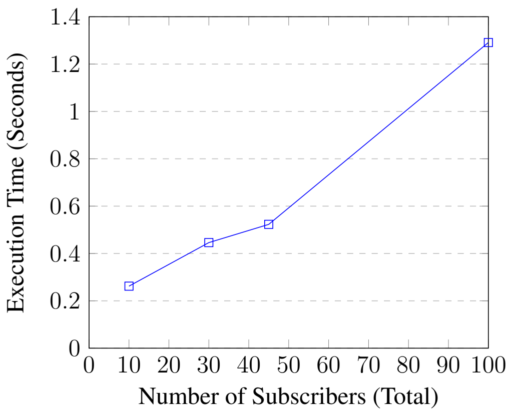
(b) UDAP [5-10].



(c) UDAP [10-10].

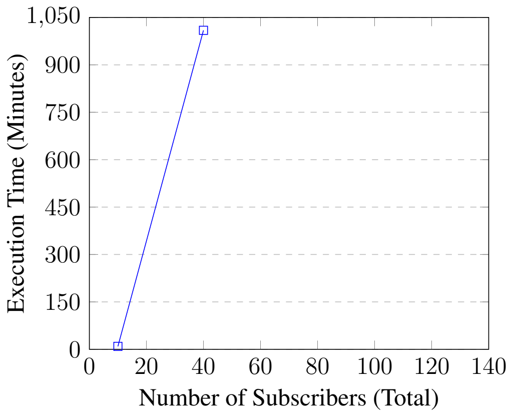


(d) Hypercycle.

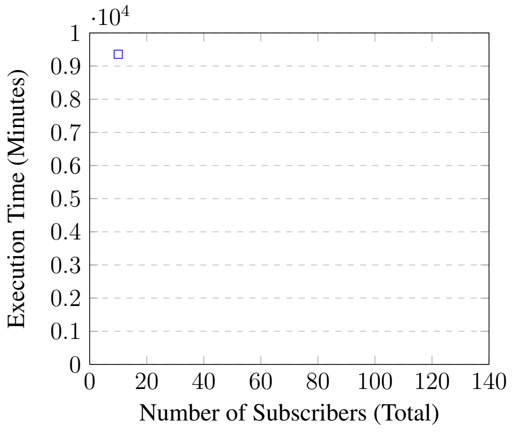


(e) Microcycle.

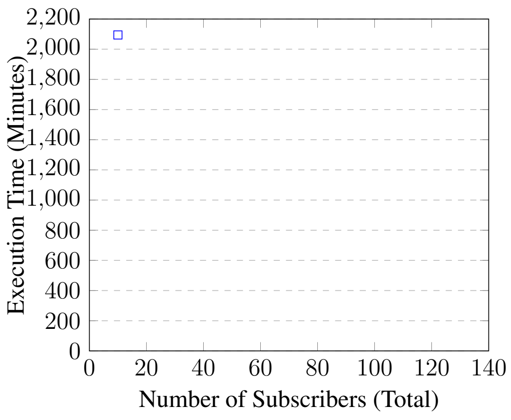
Figure 5.10: Execution time per number of subscribers on scenarios using only large flows.



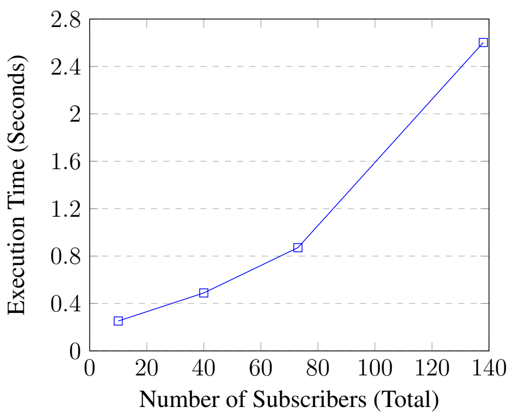
(a) UDAP [5-5].



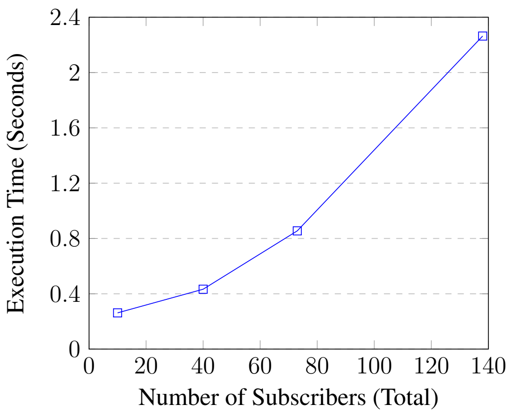
(b) UDAP [5-10].



(c) UDAP [10-10].



(d) Hypercycle.



(e) Microcycle.

the weight of the number of switches in the path as for increasing the number of formulas handled by the solver.

Finally, we can better visualize this hindrance caused by the scheduling of additional packets and cycles in Table 5.5. In this table, we see that the UDAP[10-10] approach had the worst overall performance, as a reflection of the scalability issues of the scheduling problem, followed by the UDAP[5-10] and the UDAP[5-5] approaches. The Hypercycle and Microcycle approaches had similar performance overall. We mark with TO the experiments for which TSNSCHED did not return a result within 80 hours and NA if the experiment was not performed.

As for the second batch of experiments comparing the Hypercycle and Microcycle approaches, the execution times from scenarios A to E (flows with closely related periods) can be seen in Table 5.6, and the execution times from scenarios F to I (flows with non-closely related periods) can be seen in Table 5.7. We indicate the experiments that were killed by the operational system due to excessive memory consumption as MO.

As previously discussed, the Microcycle approach is capable of providing results on a faster pace when compared to the Hypercycle approach, still, it is interesting to take in consideration the network requisites regarding the usage of bandwidth, where the former might not be adequate for the desired scenarios. The difference between the performance of the two approaches is notable on Case F, shown in Table 5.7, where the factor between the two execution times is of almost 175, displaying the weigh that comes with the usage of a higher amount of transmission windows.

Finally, it is important to highlight the Cases G, H and I, where TSNSCHED was not capable of generate schedules for the scenarios as it was terminated by the operational system. The tool was terminated within the first 24 hours of execution after depleting the 16 gigabytes of RAM dedicated to the machine. This demonstrates the impact of the complexity of the problem regarding the resources needed to formulate the scheduling problem, as the size of the flattened constraints (*i.e.*, using conjunctions to implement the constraints instead of using quantifiers) become exceedingly high.

Table 5.5: Execution time (in minutes) of three different scheduling approaches varying with the size and number of the scheduled flows.

Flow	**Small Flows**				**Medium Flows**				**Large Flows**			
Size	**1**	**3**	**5**	**10**	**1**	**3**	**5**	**10**	**1**	**3**	**5**	**10**
UDAP [5-5]	0.32	5.63	222.03	1163.07	3.36	98.02	1159.52	TO	9.29	1009.259	TO	NA
UDAP [5-10]	5.31	50.03	585	TO	58.11	3170.56	TO	NA	155.93	TO	NA	NA
UDAP [10-10]	25.72	TO	NA	NA	379.3	TO	NA	NA	2094.55	TO	NA	NA
Hypercycle	0.003	0.004	0.005	0.012	0.003	0.007	0.009	0.021	0.004	0.008	0.016	0.043
Microcycle	0.003	0.004	0.006	0.012	0.004	0.007	0.008	0.021	0.004	0.007	0.014	0.037

Table 5.6: Execution time (in seconds) for cases A to E from the batch number 2.

	Case A	Case B	Case C	Case D	Case E
Hipercycle	0.197	0.319	0.413	2.082	92.288
Microcycle	0.199	0.241	0.318	0.573	2.334

Table 5.7: Execution time (in seconds) for cases F to I from the batch number 2.

	Case F	Case G	Case H	Case I
Hipercycle	8740.63	MO	MO	MO
Microcycle	51.35	MO	MO	MO

Results and Analysis

The summarized results for the first batch of experiments is shown in Tables 5.8, 5.9, 5.10, 5.11 and 5.12 and are expressed as jit / lat, where jit is the average jitter and lat is the average latency. Also for the tables shown in this section, we mark with TO the experiments for which TSNSCHED did not return a result within 80 hours and NA if the experiment was not performed. If a scenario exceeds the maximum execution time, the scenarios with the same scheduling approach, same flow size and bigger number of flows to schedule are not executed, as it is expected that they take longer to run.

For the batch number 1, overall, TSNSCHED performed well. In all approaches, when not hindered by the execution time, it was able to construct schedules that satisfy the network performance criteria (maximum latency of 1000μs per packet and maximum jitter of 25μs per packet per flow) for networks with up to 138 subscribers and up to 10 publishers. We were also positively surprised by the overall average jitter which in most cases was well below the maximal admissible jitter. Even in the larger experiments, the schedule generated could guarantee average jitter of 6.06μs (UDAP[5-10], 1 small flow). For a comparison, the average jitter in experiments with unicast flows described in [30] was between 10μs and 50μs.

We note that cases where the number of packets is higher than the number of cycles to be scheduled have a higher average latency, which can be attributed to the capability of the

Table 5.8: Summary of experimental results for the experiments with the UDAP[5-5] scheduling approach.

No of Flows	**1**	**3**	**5**	**10**
Small Only	3.86μs/52.74μs	4.51μs/47μs	1.99μs/455.98μs	5.33μs/557.87μs
Medium Only	3.30μs/59.08μs	3.60μs/92.93μs	4.13μs/316.56μs	TO
Large Only	1.80μs/72.85μs	3.17μs/189.38μs	TO	NA

Table 5.9: Summary of experimental results for the experiments with the UDAP[5-10] scheduling approach.

No of Flows	**1**	**3**	**5**	**10**
Small Only	6.06μs/948.65μs	3.89μs/365.78μs	5.11μs/733.31μs	TO
Medium Only	4.34μs/953.76μs	5.97μs/738.13μs	TO	NA
Large Only	4.03μs/637.20μs	TO	NA	NA

tool to create more flexible cycle sizes, as demonstrated by the values in Table 5.9.

Furthermore, we observe the effectiveness of the approaches with automated application periods, which are capable of generating jitterless schedules with latencies close minimal within less than a second, given the tested scenarios for this batch.

For the second batch of experiments, we observe how the interaction of flows with different periodicities can affect the overall schedule generated by TSNSCHED. For the scenarios with flows of closely related periods, variations on the latency start to appear in some cases when using the Hypercycle approach, which can be attributed to the capability of the solver to place transmission windows in arbitrary positions of a bigger cycle, so long the network requirements are met. Due to the structure of solutions following the Microcycle approach, transmission windows are in the same position of subsequent microcycles, avoiding the behavior presented by the cases where the Hypercycle approach has been used.

As for the cases with non-closely related periodicities, only one of the considered scenarios could be successfully executed by TSNSCHED. The schedule for Case F, when generated using the Hypercycle approach, had a latency of 26.13μs and 1.86μs of jitter, and when generated using the Microcycle approach, had a latency of 42.66μs and no jitter.

It is important to highlight that TSNSCHED managed to return valid schedules for all sce-

Table 5.10: Summary of experimental results for the experiments with the UDAP[10-10] scheduling approach.

No of Flows	**1**	**3**	**5**	**10**
Small Only	1.75μs/46.07μs	TO	NA	NA
Medium Only	2.52μs/58.97μs	TO	NA	NA
Large Only	1.28μs/74.78μs	TO	NA	NA

Table 5.11: Summary of experimental results for the experiments with the Hypercycle scheduling approach.

No of Flows	**1**	**3**	**5**	**10**
Small Only	0μs/48.2μs	0μs/48.33μs	0μs/47.09μs	0μs/200.51μs
Medium Only	0μs/64.1μs	0μs/73.66μs	0μs/154.58μs	0μs/319.02μs
Large Only	0μs/84.4μs	0μs/159.17μs	0μs/219.04μs	0μs/511.45μs

narios that had no issues regarding the scalability of the tool and, consequently, no problems with resources and execution time.

Table 5.12: Summary of experimental results for the experiments with the Microcycle scheduling approach.

No of Flows	**1**	**3**	**5**	**10**
Small Only	0μs/48.2μs	0μs/48.33μs	0μs/47.09μs	0μs/200.51μs
Medium Only	0μs/64.1μs	0μs/73.66μs	0μs/154.58μs	0μs/319.02μs
Large Only	0μs/84.4μs	0μs/159.17μs	0μs/219.04μs	0μs/511.45μs

Table 5.13: Execution time (in seconds) for cases A to E from the batch number 2.

	Case A	Case B	Case C	Case D	Case E
Hipercycle	0μs/34.33μs	11μs/22.33μs	2.75μs/525.98μs	6μs/16.93μs	6.29μs/21.71μs
Microcycle	0μs/34.33μs	0μs/310.66μs	0μs/11.33μs	0μs/171μs	0μs/11.33μs

Chapter 6

Closing Remarks

This dissertation describes in depth the tool TSNSCHED, the first, to the best of our knowledge, openly accessible tool that generates schedules for high performance deterministic cyclic time-sensitive networks. Moreover, we present the first experimental results for schedule generation for multicast flows, evaluating our tool on relatively large networks (up to 138 subscribers) with varying flow characteristics (number, size, amount of packets and cycles). Overall, TSNSCHED is able to generate schedules with acceptable network performance (jitter and latency).

The tools described in [30] and [10] are proprietary. Aside from this, TSNSCHED can generate schedules that comply with latency, jitter and bandwidth reservation for best-effort traffic, capable of generating schedules that may not only comply but be much more efficient than the given restrictions, as we can see from the presented experiments.

While TSNSCHED currently may only be used for offline schedule generation, its applications are countless, taking in consideration its scalability limitations. From industrial systems to smart cars and other applications directly implemented into modern society. Future works includes methods to enhance the time-efficiency of the tool, as well as rethink the constraints being used to generate the output in order to reduce the number of formulas to be solved by the schedule.

As future work, we believe that TSNSCHED's flexibility and configuration level gives us the necessary grounds to explore new approaches for the improvement of schedule generation for TSN. We also aim to implement other mechanisms which compose the TSN standards, giving a broader set of possible configurations for convergent networks.

Another option is to, with the knowledge built during the development of TSNSCHED, attempt to generate an approach of structure similar to the one presented in this dissertation, but using ILP.

Currently, we are studying the integration of TSNSCHED into frameworks for automatically configuring physical devices, as we believe that the tool already has the necessary interfaces to enable an automatic deduction of network topology and scheduling requirements, as well the adaptation of TSNSCHED in order generate outputs ready for deployment to TSN switches.

Bibliography

[1] S. Beji, S. Hamadou, A. Gherbi, and J. Mullins. Smt-based cost optimization approach for the integration of avionic functions in ima and ttethernet architectures. In *Proceedings of the 2014 IEEE/ACM 18th International Symposium on Distributed Simulation and Real Time Applications*, pages 165–174. IEEE Computer Society, 2014.

[2] Belden/Hirschmann. *Time-Sensitive Networking For Dummies - Belden/Hirschmann Special Edition*. John Wiley and Sons, Inc., 2018.

[3] B. Brandenbourger and F. Durand. Design pattern for decomposition or aggregation of automation systems into hierarchy levels. In *2018 IEEE 23rd International Conference on Emerging Technologies and Factory Automation (ETFA)*, volume 1, pages 895–901. IEEE, 2018.

[4] Cisco. Time-sensitive networking: A technical introduction. `https://www.cisco.com/c/dam/en/us/solutions/collateral/industry-solutions/white-paper-c11-738950.pdf`, 2017. Accessed: 2020-01-02.

[5] I. I. Consortium et al. Time sensitive networks for flexible manufacturing testbed-description of converged traffic types. Technical report, Tech. Rep, 2018.

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

[7] S. S. Craciunas, R. S. Oliver, and T. C. AG. An overview of scheduling mechanisms for time-sensitive networks. *Proceedings of the Real-time summer school LÉcole dÉté Temps Réel (ETR)*, 2017.

[8] S. S. Craciunas, R. S. Oliver, M. Chmelík, and W. Steiner. Scheduling real-time communication in ieee 802.1 qbv time sensitive networks. In *Proceedings of the 24th International Conference on Real-Time Networks and Systems*, pages 183–192. ACM, 2016.

[9] S. S. Craciunas, R. S. Oliver, and W. Steiner. Formal scheduling constraints for time-sensitive networks. *arXiv preprint arXiv:1712.02246*, 2017.

[10] S. S. Craciunas, R. S. Oliver, and W. Steiner. Demo abstract: Slate xns–an online management tool for deterministic tsn networks. In *2018 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 103–104. IEEE, 2018.

[11] L. De Moura and N. Bjørner. Z3: An efficient smt solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008.

[12] A. C. T. dos Santos, B. Schneider, and V. Nigam. Tsnsched. `https://github.com/ACassimiro/TSNsched`, 2019. Accessed: 2019-04-29.

[13] F. Dürr and N. G. Nayak. No-wait packet scheduling for ieee time-sensitive networks (tsn). In *Proceedings of the 24th International Conference on Real-Time Networks and Systems*, pages 203–212. ACM, 2016.

[14] J. Falk, F. Dürr, and K. Rothermel. Exploring practical limitations of joint routing and scheduling for tsn with ilp. In *2018 IEEE 24th International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, pages 136–146. IEEE, 2018.

[15] M. H. Farzaneh, S. Kugele, and A. Knoll. A graphical modeling tool supporting automated schedule synthesis for time-sensitive networking. In *2017 22nd IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*, pages 1–8. IEEE, 2017.

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

[17] J. Happich. Time Sensitive Networks approaches the car, 2016. Available online at `https://www.eenewsautomotive.com/news/time-sensitive-networks-approaches-car` last accessed on May 22th 2019.

[18] R. Hummen, S. Kehrer, and O. Kleineberg. White paper: Tsn-time sensitive networking. *Belden, St. Louis, MI, USA, Tech. Rep*, 2017.

[19] IEEE. IEEE 802.1ASRev - Timing and Synchronization for Time-Sensitive Applications. Available online at `https://1.ieee802.org/tsn/802-1as-rev/` last accessed on March 30th 2018.

[20] IEEE. IEEE 802.1Qbv - Enhancements for Scheduled Traffic. Available online at `http://www.ieee802.org/1/pages/802.1bv.html` last accessed on March 30th 2018.

[21] IEEE. IEEE 802.1Qci – Per-Stream Filtering and Policing, 2016. Available online at `https://1.ieee802.org/tsn/802-1qci/` last accessed on April 15th 2019.

[22] R. S. Oliver, S. S. Craciunas, and W. Steiner. Ieee 802.1 qbv gate control list synthesis using array theory encoding. In *2018 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 13–24. IEEE, 2018.

[23] L. C. Paulson. The foundation of a generic theorem prover. *Journal of Automated Reasoning*, 5(3):363–397, 1989.

[24] F. Pozo, W. Steiner, G. Rodriguez-Navas, and H. Hansson. A decomposition approach for smt-based schedule synthesis for time-triggered networks. In *2015 IEEE 20th conference on emerging technologies & factory automation (ETFA)*, pages 1–8. IEEE, 2015.

[25] Renesas. In-Vehicle Networking Solutions, 2019. Available online at `https://www.renesas.com/in/en/solutions/automotive/technology/networking-solutions.html` last accessed on May 22th 2019.

[26] S. Roy Chowdhury. Packet scheduling algorithms for a software-defined manufacturing environment, 2015.

[27] J. Specht and S. Samii. Urgency-based scheduler for time-sensitive switched ethernet networks. In *2016 28th Euromicro Conference on Real-Time Systems (ECRTS)*, pages 75–85. IEEE, 2016.

[28] J. Specht and S. Samii. Synthesis of queue and priority assignment for asynchronous traffic shaping in switched ethernet. In *2017 IEEE Real-Time Systems Symposium (RTSS)*, pages 178–187. IEEE, 2017.

[29] W. Steiner. An evaluation of smt-based schedule synthesis for time-triggered multi-hop networks. In *2010 31st IEEE Real-Time Systems Symposium*, pages 375–384. IEEE, 2010.

[30] W. Steiner, S. S. Craciunas, and R. S. Oliver. Traffic planning for time-sensitive communication. *IEEE Communications Standards Magazine*, 2(2):42–47, 2018.

[31] S. Thangamuthu, N. Concer, P. J. Cuijpers, and J. J. Lukkien. Analysis of ethernet-switch traffic shapers for in-vehicle networking applications. In *Proceedings of the 2015 Design, Automation & Test in Europe Conference & Exhibition*, pages 55–60. EDA Consortium, 2015.

[32] TTTech. Ieee tsn (time-sensitive networking): A deterministic ethernet standard. `http://www.hit.bme.hu/~jakab/edu/litr/TimeSensNet/TTTech_Time-Sensitive_Networking_Whitepaper.pdf`, 2015. Accessed: 2020-01-02.

[33] Z3Prover. The z3 theorem prover. `https://github.com/Z3Prover/z3`, 2019. Accessed: 2019-03-04.

[34] M. Zeller and C. Prehofer. Modeling and efficient solving of extra-functional properties for adaptation in networked embedded real-time systems. *Journal of Systems Architecture - Embedded Systems Design*, 59(10-C):1067–1082, 2013.

Appendix A

TSNSCHED's Architecture and Classes

A.1 Overview of Classes

Being aware of the classes presented in figure 5.1, their functionalities and quirks are shortly described bellow.

A.1.1 Device

The Device class represents a start or end device in a TSN flow. All the properties of device nodes in the network are specified here. These properties are rather trivial to understand but they build the core of the constraints of a flow. Here, the user can specify the packet periodicity of the flow and the hard constraint (maximum allowed latency).

A.1.2 Flow

This class specifies a flow (or a stream, in other words) of packets from one source to one or multiple destinations. It contains references for all the data related to this flow, including path, timing, packet properties, so on and so forth. The flows can be unicast type or Publish/Subscribe type flows.

In a more in depth perspective, each flow will later be broken into "smaller flows", called flow fragments. This class will also store the reference to them in a PathTree object.

A.1.3 FlowFragment

This class is used to represent a fragment of a flow. Simply put, a flow fragment represents the flow it belongs to regarding a specific switch in the path. With this approach, a flow, regardless of its type, can be broken into flow fragments and distributed to the switches in the network. It holds the values of the departure time (leaving the previous node), arrival time (arriving in the current node) and scheduled time (leaving the current node) of packets from this flow on the switch it belongs to. Since these times are specified as Z3 objects, there is no need to store copies of them, just the reference.

This approach allows the user to have a more encapsulated code, since it doesn't matter the type of flow being used here. The tool can simply break the flow of packets into streams that only transfer data over one switch device (cover only one hop), and visualize this stream (the fragment) as a link in a chain composing the whole path of switches.

FlowFragment objects store references to the index of the current and next switch nodes in the flow's path and also the departure time, arrival time, scheduled time of the packets that go through it. It is important to have in mind that the timing variables stored by FlowFragment objects are float values, not Z3 variables. The Z3 variables for these values can be retrieved through the port or the switch that this fragment goes through.

A.1.4 Switch

Contains most of the properties of a normal switch that are used to build the schedule. Since this scheduler doesn't take in consideration scenarios where normal switches and TSN switches interact, no Z3 properties had to be specified in this class.

It is currently used as parent class for TSNSwitch. Can be used to further extend the usability of this project in the future.

A.1.5 TSNSwitch

This class contains the information needed to specify a switch capable of complying with the TSN patterns to the schedule. Aside from containing part of the Z3 data used to generate the schedule, objects created from this class are able to organize a sequence of ports that connect the switch to other nodes in the network. TSNSwitch objects also have reference the

Z3 variables for the departure, arrival and scheduled time of FlowFragment objects stored in its ports.

A.1.6 Network

Using this class, the user can specify the network topology using switches and flows. The network will be given to the scheduler generator so it can iterate over the network's flows and switches setting up the scheduling rules.

In the current implementation, the upper bound jitter variation is specified in the Network, making it uniform for all flows added to the topology.

A.1.7 PathNode

In a Publish/Subscribe flow, contains the data needed in each node of a pathTree. Since a Publish/Subscribe flow path can be seen as a tree, a single node on that tree is a PathNode.

It has a reference to its father, possesses an device or TSNSwitch, a list of children and a reference to a flow fragment for each of the children in case of being a switch.

A.1.8 PathTree

Used to specify the path on Publish/Subscribe flows. Has a reference to the PathNode root, which contains the starting device, and also references to the leaves, which contain the destinations of the publisher messages.

In simple words, it is a tree of PathNodes with a few classic tree methods.

A.1.9 Cycle

The Cycle class represents the cycle of a TSN switch. A cycle is a time interval with a specific duration where time windows can be distributed according to constraints to prioritize critical traffic. During these time windows, the gate of the respective priority queue will be open. Each cycle has a a start, a duration and a set of sequence of time windows (a sequence of priority slots for each priority).

After the specification of its properties through user input, the *toZ3* method can be used to convert the values to Z3 variables and query the unknown values.

There is no direct reference from a cycle to its time slots. The user must use a priority from a flow to reference the time window of a cycle. This happens because of the generation of Z3 variables.

For example, if I want to know the duration of the time slot reserved for the priority 3, it most likely will return a value different from the actual time slot that a flow is making use. This happens due to the way that Z3 variables are generated. A flow fragment can have a priority 3 on this cycle, but its variable name will be "flowNfragmentMpriority". Even if Z3 says that this variable's value is 3, the reference to the cycle duration will be called "cycleXSlotflowNfragmentMpriorityDuration", which is clearly different from "cycleXSlot3Duration".

To make this work, every flow that has the same time window has the same priority value. And this value is limited to a maximum value numOfSlots. So, to access the slot start and duration of a certain priority, a flow fragment that uses the priority must be retrieved. This also deals with the problem of having unused priorities, which can end up causing problems due to constraints of guard band and such.

A.1.10 Port

This class is used to implement the logical role of a port of a switch for the scheduler. The core of the scheduling process happens here. Simplifying the whole process, the other classes in this project are used to create, manage and break flows into smaller pieces. These pieces are given to the switches, and from the switches they will be given to their respective ports according to the path of the flow.

After this is done, each port now has an array of fragments of flows that are going through them. This way, it is easier to schedule the packets since all you have to focus are the flow fragments that might conflict in this specific port. The type of flow, its path or anything else does not matter at this point.

A.1.11 ScheduleGenerator

Used to generate a schedule based on the properties of a given network through the method generateSchedule. Will create a log file and store the timing properties on the cycles and flows.