Fault Injection and Reliability Analysis on Time Sensitive Networking

Renan Moreira da Silva

CENTRO DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM INFORMÁTICA
UNIVERSIDADE FEDERAL DA PARAÍBA

João Pessoa, 2024

Catalogação na publicação
Seção de Catalogação e Classificação

S586f Silva, Renan Moreira da.
Fault injection and reliability analysis on time sensitive networking / Renan Moreira da Silva. - João Pessoa, 2024.
57 f. : il.

Orientação: Vivek Nigam.
Coorientação: Iguatemi Fonseca.
Dissertação (Mestrado) - UFPB/CI.

1. Informática. 2. Rede sensível ao tempo (TSN). 3. Simulação. 4. Tolerância a falhas. I. Nigam, Vivek. II. Fonseca, Iguatemi. III. Título.

UFPB/BC CDU 004(043)

Elaborado por Larissa Silva Oliveira de Mesquita - CRB-15/746

Renan Moreira da Silva

Fault Injection and Reliability Analysis on Time Sensitive Networking

Manuscript submitted to the Coordination of the Postgraduate Course in Informatics at the Federal University of Paraíba as part of the requirements necessary to obtain the Master's degree in Informatics

Advisors: Vivek Nigam and Iguatemi Eduardo da Fonseca

July 2024

Abstract

Time Sensitive Networking (TSN) provides high performance communication using time scheduling to achieve deterministic latency and jitter. In theory, the rigor of a TSN schedule is the key to achieve deterministic communication. However, real devices are prone to errors. TSN-based applications require both fault-tolerance and end-to-end latency guarantee. In this work we identify the limitations of the TSN scheduling method, enhancing the simulation model NeSTiNg to enable fault-injection. The fault-injection is performed by introducing new components, and modules to emulate permanent, transient and intermittent faults based on probability distributions. We evaluate the effectiveness of the fault-injection comparing the latency and jitter results to the ones expected by the schedule generator TSNsched, also presenting a technique to increase network reliability on faulty scenarios. Moreover, we analyze the the schedule generated by TSNsched using criteria such as path and transmission window consistency. In conclusion, we were able to prove the effectiveness of the fault-tolerant techniques found on the state-of-the-art, compiling all the results and impacts on scheduled and non-critical traffic.

Keywords: TSN, Simulation, Fault-Tolerance.

List of Figures

List of Tables

List of Acronyms

TSN – Time Sensitive Networking

TAS – Time-Aware Shaper

AVB – Audio-Video Bridging

PTP – Precision Time Protocol

TDMA – Time-Division Multiple Access

NTP - Network Time Protocol

PCP – Priority Code Point

FIFO – First In First Out

GCL – Gate Control List

FRER – Frame Replication and Elimination for Reliability

SMT – Satisfiability Modulo Theories

LCM – Least Common Multiple

GCD – Greatest Common Divisor

ILP – Integer Linear Programming

IDE – Integrated Development Environment

UML – Unified Modeling Language

RSPC – Runtime Safety Properties Checker

JSON – JavaScript Object Notation

Contents

1 Introduction

Modern industrial systems have several traffic types, such as, time-critical, best-effort, and audio/video. Each of these traffic flows has different priorities and requirements making the network design a challenging task [1]. An increasingly diverse range of fields of the industry require deterministic latency and jitter in their communication, including automotive, aviation, and smart factories. Thus, new methods for network design and communication need to be applied [2].

Time-Sensitive Networking (TSN) is a set of standards developed by IEEE 802.1 Task Group [10] that enable the co-existence of mixed traffic classes on a network. The main objective of TSN is to achieve deterministic communication with low latency and jitter. To achieve this, TSN shapes the network traffic using: time synchronization (IEEE 802.1ASRev), packet preemption (IEEE 802.1Qbu), traffic scheduling through time (IEEE 802.1Qbv) [4] and other standards that are explained in details in Chapter 2.

The Time-Aware Shaper (TAS) is the scheduling technique proposed by the TSN standard IEEE 802.1Qbv. The generation of a TAS schedule for the network switches is critical to achieve the deterministic communication proposed by TSN. However this task is challenging since several factors should be considered, such as packet size, periodicity, number of devices, route, link capacity, latency and jitter. Thus, the TAS scheduling an NP-complete problem [5].

The generation of a TAS schedule in an optimal manner has been studied for some time. As later explained in Chapter 3, among the studies found on the state of the art, TSNsched [6] is highlighted as one of the best open source tools to generate schedules, supporting unicast and multicast flows, such as, in Publish/Subscribe networks.

To create a schedule, the network details must be specified in advance, so the TAS can calculate and generate the network schedule for each traffic class. Since all schedule are pre-calculated, static values are used to represent relevant parameters, such as packet size, periodicity, number of devices, route, transmission delay and processing delay.

However, when we work with real scenarios, devices may not work in a constant stable manner, presenting errors and inconsistencies. If an external factor such as interference, faulty device or link failure occurs, the traffic would be affected and the computed schedule for that network would become irrelevant. By the way, tools like TSNsched can be seen as black-boxes, and since they contain thousands of lines of code and external tools, it may not be clear whether their output can be trusted or not.

Fault-tolerant aspects must be taken into consideration when creating a TAS schedule. Since TSN is a combination of standards, we can find fault-tolerant mechanisms on the base standards. Among them, we highlight the redundancy management standard

IEEE 802.1CB, the flow control and reservation IEEE 802.1Qca, and the flow filtering and policing IEEE 802.1Qci. These three standards work together to decrease the latency and packet loss by duplicating packets and sending them in alternative routes, bearing in mind the elimination of duplicated frames at the destination [7]. However these standards have limitations. All relevant TSN standards are explained in details in Chapter 2.

NeSTiNg [8] is a simulation model for OMNeT++, using the INET-Framework and enhancing it adding components to enable TSN simulation. In this work we further enhance this simulation model to enable fault-injection in the network. Thus, allowing us to analyze the network behavior and plan the next steps to achieve the balance between deterministic communication and fault-tolerance.

1.1 Research Proposal

TSN have a set of standards that must be followed to achieve deterministic communication. Most of these standards are related to the schedule generation, leaving the fault-tolerance as a secondary topic. The strictness of the TSN standards are a tradeoff between deterministic communication and fault-tolerance.

The goal of this work is to produce a new methodology for TSN scheduling, designing new methods to recover and prevent network faults, increasing the schedule fault-tolerance. To achieve this, we are going to identify and evaluate the robustness limits of TSN scheduling stress scenarios, analyzing the solutions found on the state of the art. By the way, we use the aforementioned schedule generation tool TSNsched, and the NeSTiNg simulation model.

TSNsched was developed to generate schedules according to the TAS (IEEE 802.1Qbv), but the fault-tolerant aspects of TSN were not taken in consideration during the tool development. This work aims to enhance the schedule generator, adding concepts of fault-tolerance, enabling the generation of more robust schedules. We also propose a methodology to validate the output given by TSNsched, checking for errors and consistency of the schedule generated.

Furthermore, we also aim to enhance the simulation model NeSTiNg to enable fault-injection to allow the simulation of more diverse and realistic scenarios. In addition, the validation and execution of the simulation with fault-injection will be added to the TSNsched tool-chain.

1.1.1 General Objectives

Enhance the capabilities of NeSTiNg and TSNsched by incorporating fault-injection into simulations and integrating fault-tolerance mechanisms into schedule generation. Au-

tomate this process and integrate it into the TSNsched tool-chain, while also developing novel methods for evaluating and generating resilient network schedules.

1.1.2 Specific Objectives

The specific objectives of this work are:

- Create new modules on NeSTiNg to enable fault-injection.
- Add new mechanisms to TSNsched to achieve fault-tolerance.
- Evaluate the schedule generated by TSNsched using simulation tools.
- Quantify the improvement of the faulty network with the fault-tolerant mechanisms.
- Automate the fault-injection on the TSNsched tool-chain.

1.2 Dissertation Structure

This dissertation is organized as follows:

Chapter 2 gives an overview of the TSN sub-standards, highlighting the scheduling mechanisms, the fault-tolerant standards and tools chosen.

In Chapter 3, we explore tools and models relevant to TSN scheduling, simulation, and fault-injection. This exploration is based in the contemporary state of the art, providing an overview of the advances and methodologies of TSN scheduling and its associated components.

Chapter 4 presents details about the tools enhancement. Explaining the selected faults, as well as its simulation strategies. Additionally, we introduce a set of criteria to validate the generated schedule, as well as, present a method to improve the schedule generation.

In Chapter 5 we analyze the results of the network schedule with and without fault injection. Comparing these results and adding fault-tolerant techniques to the proposed scenario.

Finally, Chapter 6 is devoted to the conclusions of this work.

2 Theoretical Foundation

This chapter is dedicated to the TSN standards. And discusses concepts of Time Sensitive Scheduling, Fault-tolerance and the tools used in this work, such as TSNsched and NeSTiNg.

2.1 Time Sensitive Networking

In order to understand TSN, it is necessary to introduce its predecessor. Audio-Video Bridging (AVB) [9] is a set of standards developed to enable the transmission of multimedia data, such as video and audio, constantly and without interruptions. The development group (IEEE 802.1 Task Group [10]) responsible for the AVB was renamed as TSN Task Group [11] to reflect the expanding scope of its work, which is to provide deterministic real-time communication over Ethernet.

Deterministic Ethernet refers to networked communication technology that uses time scheduling to bring deterministic real-time communication to standard IEEE 802.3 Ethernet. This can be achieve using a global sense of time and a schedule which is shared among network components [12]. TSN offers a way to send time-critical traffic over a standard Ethernet infrastructure, enabling the convergence of all traffic classes and multiple applications in one network.

The purpose of TSN is to allow end-to-end deterministic communication, that is, TSN seeks to enable the transmission of packets with latency and jitter restrictions. As the latency is the delay in the delivery of the packages of a network and the jitter is the statistical variation of this delay. In order to keep this delay in sending packets under a certain limit, TSN makes use of several mechanisms in the form of standards. The main TSN standards can be seen in Table 1.

Table 1: Subset of TSN standards. Adapted from [4].

Standard	Description
802.1Qbv	Time-aware Shaping (queue based)
802.1ASrev	Timing and Synchronization
802.1CB	Redundancy (replication and elimination of packets)
802.1Qca	Flow control and reservation
802.1Qbu	Packet preemption
802.1Qcc	Stream reservation improvements
802.1Qch	Queuing and Cyclical Sending
802.1Qci	Flow filtering and policing

As the core concept of TSN, IEEE 802.1Qbv defines a mechanism to control the flow on TSN switches using priority queues and time windows. The schedule created by

this standard is distributed among the participating network devices. One of the main artifacts of this standard is the Time-Aware Shaper (TAS), which will be discussed in details in the following Section 2.2.

To achieve deterministic communication, timing and synchronization are crucial. The IEEE 802.1ASrev is a revision of the previous IEEE 802.1AS standard adding capabilities of the IEEE 1588 PTP (Precision Time Protocol). The standard IEEE 1588 uses an algorithm called Best Master Clock. This algorithm is used to determine which of the devices that are part of the network has the most accurate clock, which will be used as a reference for other devices (Grand Master Clock).

Figure 1 depicts one important mechanism of TSN synchronization, the Time-Division Multiple Access (TDMA). This technique allows the division of time in cycles of same length, and time-slots often called send-window. Thus, the network switches can work in a periodic behavior having the same concept of time.

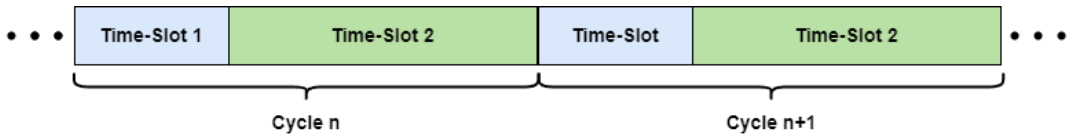

Figure 1: Time division in cycles and time-slots.

Additionally, IEEE 1588 provides support to multiple time domains that can be synchronized in parallel. The network participants can be synchronized with a global time reference, such as NTP (Network Time Protocol), as well as a secondary network time reference. This gives the option to use the global synchronization (NTP) for event logging, and the network-wide synchronized clock (PTP) can be used for scheduling [13].

The IEEE 802.1CB standard is responsible for the redundancy management in TSN. This standard is responsible to send redundant copies of a message. In other words, the message is duplicated and sent in parallel through different paths. The goal of this standard is to increase the availability, mitigating the information loss caused by network faults. To avoid network overload, the duplicated messages can be eliminated at a intermediate switch or at an end-device. The IEEE 802.1Qca standard is responsible to set up the disjoint paths used by the redundancy management mechanism.

The frame preemption is done by the standard IEEE 802.1Qbu. Frame preemption is the capability to stop the transmission of a big packet for a brief period of time, splitting it into two smaller parts and sending the remaining information later. TSN uses this standard to optimize bandwidth usage, allow transmission of packets with a higher priority, in order to prevent delays.

As depicted in Figure 2, the transmission of a packet with a lower priority must be interrupted withing a range, so it cannot affect the next transmission window. This

period where no packet is allowed to be transmitted is called Guard Band, and is achieved by the *hold and release* mechanism of the standard IEEE802.3br [14].

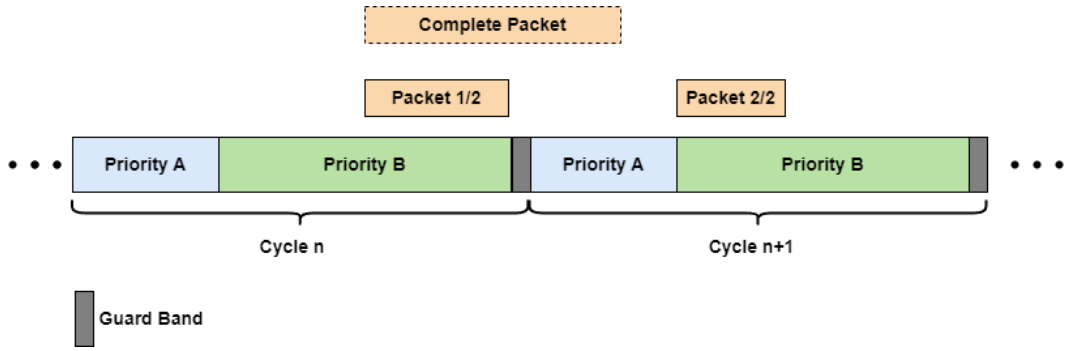

Figure 2: Cycle view for packet preemption.

2.2 Time-Aware Shaper

As stated previously, the purpose of TSN is to allow end-to-end deterministic communication, providing transmission with latency and jitter restrictions. In order to keep this delay under a certain limit, TSN makes use of several mechanisms in the form of standards. One of the main components of TSN is the Time-Aware Shaper (TAS) [15], this mechanism is responsible for scheduling traffic into eight different priorities. Figure 3 illustrates how this mechanism works.

Computing a correct and coordinated TAS schedule is an NP-complete problem given the amount of factors and restrictions that need to be taken consideration, such as packet size, periodicity, number of devices, route, link capacity, latency, jitter [5].

TSN packets have a VLAN identifier on the IEEE 802.1Q header [16]. This identifier is called Priority Code Point (PCP), and as the name suggest, determines that packet priority on the moment of creation. Each port on a TSN switch have up to eight priority queues. When a packet arrives at a switch, it is processed and goes through a filter, often called the *Switching Fabric*. This Priority Filter will check the PCP of the incoming packet and direct it to the corresponding priority queue.

Each Priority Queue may have a different transmission selection algorithm associated to it. This algorithm will determine the next package and if it can be transmitted. The most common transmission selection algorithms are:

- **Credit-Based Shaper** [17]: The algorithm defines credits associated to each priority queue. The transmission is only allowed when the credits of the queue are greater or equal to zero. When a packet of that queue is sent, the credits are decreased, and when the queue remains inactive, the credits are increased.

- **Strict Priority**: This algorithm is used as the default in most applications. It simply uses a FIFO (First In First Out) policy on the active queue.

In addition to the transmission selection algorithm, each queue have a Forwarding Gate. This gate have two states: open (1) and closed (0). The packets of that queue can only be transmitted if the respective forwarding gate is open. What determines the status of the forwarding gate is the scheduling table called Gate Control List (GCL).

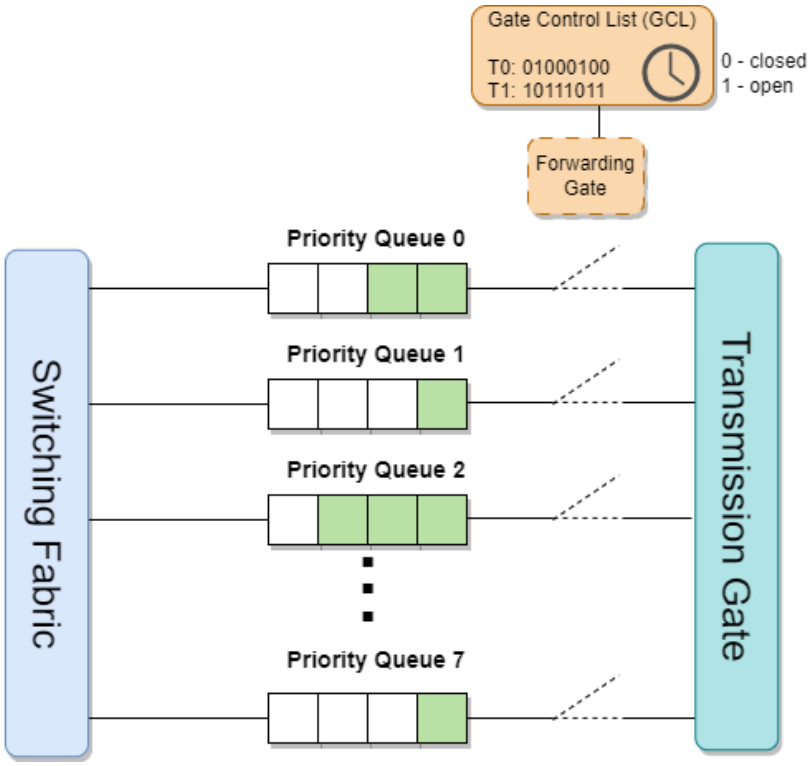

Figure 3: Time-Aware Shaper.

Using the time synchronization and TDMA from the IEEE 802.1ASrev standard, is possible to give a priority to the Time-Slots. Figure 4 depicts a cycle with two time-slots, with priority A and B. During time-slot 1, only packets with Priority A can be sent, so this reservation and time-slot change can be associated to the GCL.

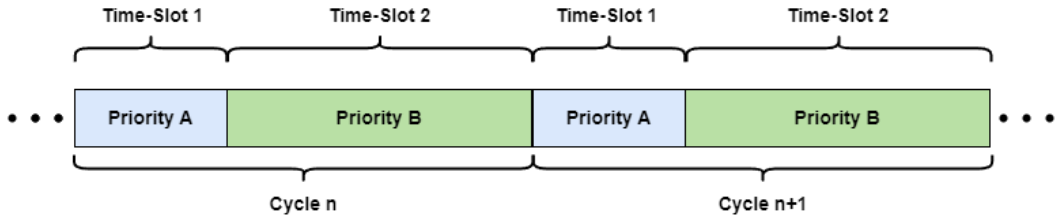

Figure 4: Priority assignment to time-slots.

2.3 Fault-Tolerance on TSN

The main goal of TSN is to provide high performance deterministic communication in a variety of applications, such as automotive and smart factories. One of the factors contributing to the high demand for reliability in TSN, is due to the capability to convergence different traffic classes and multiple applications in one network.

The robustness of a TAS schedule can be determined by the network performance when facing potential hazards. The schedule should be built to prevent faults such as link failure, frame drop and interference; and in case these faults happen, the schedule should enable a fast recovery. Parameters like bandwidth usage, latency, jitter, device connections, and efficiency serve as metrics for evaluating the robustness of the schedule. This ensures a schedule that not only anticipates challenges but also responds to them, reinforcing the network reliability and stability.

As mentioned previously, TSN have sub-standards dedicated to achieve reliability. The first standard that comes to mind for fault-tolerance on TSN is the redundancy management standard IEEE 802.1CB. The main mechanism of this is standard is called Frame Replication and Elimination for Reliability (FRER). Frames are duplicated and sent through disjoint paths to increase reliability, in the case were the network presents a fault or a device stops working, the chances of the information arriving at the destination are drastically increased with the use of this mechanism [7]. Spacial redundancy is a technique used to ensure data transmission on faulty networks, this spacial redundancy is achieved on TSN by the FRER mechanism.

However, this duplication of frames have a cost and can overload the network. Thus, the FRER works to eliminate the unnecessary duplicates, avoiding waste of resources. Even though the replicated frames take different routes, the user or main controller is not able to set this route statically. The standard responsible for this is the Flow Control and Reservation standard IEEE 802.1Qca [18].

Another standard related to fault-tolerance is the IEEE 802.1Qci [19]. This standard is responsible for the flow filtering and policing. these policing and filtering functions include the detection and mitigation of disruptive transmissions, protecting the devices from flooding with frames.

Based on this, TSN have fault-tolerant standards. However, all these standards revolve around frame replication and elimination. Stronger mechanisms should be implemented to achieve a reliable communication. So, in the subsequent chapters, we go further on the possible faults the network could face and the impacts caused on the schedule.

2.4 TSNsched

As briefly explained before, TSN have several standards and complex functionalities, making the TAS scheduling a NP-complete problem. The creation of such a schedule is a complicated task, whose difficulty grows exponentially with the size of the network and number data flows.

TSNsched [6] is an application that uses the Z3 SMT (Satisfiability Modulo Theories) Solver [20] to generate a TAS schedule given an input file with the network topology. This input file must contain information about the devices, and data flows of the network, such as: packet size, periodicity, latency and jitter requirements, etc. Figure 5 shows a fragment of the input file, containing the basic in formations for the devices and switch of a simple topology.

```
ScheduleGenerator scheduleGenerator = new ScheduleGenerator(false);
scheduleGenerator.setGenerateSimulationFiles(true);

/*
 * GENERATING DEVICES
 */
Device dev0 = new Device(500, 0, 1000, 1250);
Device dev1 = new Device(500, 0, 1000, 1250);

/*
 * GENERATING SWITCHES
 */
TSNSwitch switch0 = new TSNSwitch("switch0",100, 8, 125, 1, 400, 3000);
```

Figure 5: TSNsched input file fragment.

TSNsched is capable of synthesize IEEE 802.1Qbv schedules with latency lower than 1000 μs and jitter lower than 25 μs, for topologies *Publish/Subscribe* up to 73 subscribers and 10 multicast dataflows. This is achieved due to a set of constraints imposed to the SMT solver.

In general, SMT Solver defines a mathematical formula that may or may not be satisfied. If an answer can be found for the defined problem, it is said the satisfiability was reached. Z3 reaches this answer imposing a set of constraints to the formula. TSNsched tries to achieve a feasible schedule for the given topology using a set of constraints. The constraint seen in Figure 6 ensures that the time-slots are inside the cycle.

```
// A slot will be somewhere between 0 and the end of the cycle minus its duration (Slot in cycle constraint)
solver.add(ctx.mkGe(cycle.slotStartZ3(ctx, flowPriority, indexZ3), ctx.mkInt(0)));
solver.add(
    ctx.mkLe(cycle.slotStartZ3(ctx, flowPriority, indexZ3),
        ctx.mkSub(
            cycle.getCycleDurationZ3(),
            cycle.slotDurationZ3(ctx, flowPriority, indexZ3)
        )
    )
);
```

Figure 6: TSNsched packet scheduling constraint.

TSNsched have two main approaches for schedule generation, Hypercycle and Microcycle. When a topology have data flows with different periodicities, the size of the cycle must be adjusted to fit all the time-slots [21]. This adjustment results in a Hypercycle with size equals to the Least Common Multiple (LCM) of all periodicities.

However, the GCL of the switch may become large if many time-slots are put in a single cycle, creating a resource issue. In this case, the Microcycle approach can be used. The cycle size in this approach is equal to the Greatest Common Divisor (GCD) of all periodicities.

Figure 7 gives a graphical representation of both approaches. If we have two flows with periodicity of 200 and 700 μs, respectively. The resulting Hypercycle would have 1400 μs, while the Microcycle would have 100 μs.

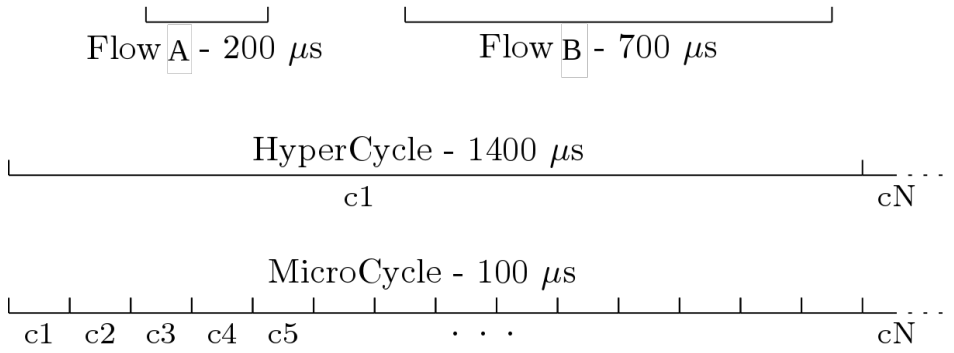

Figure 7: Hypercycles and Microcycles comparison.

Nonetheless, a key problem in using such tools is that they can be seen as black-boxes and since they contain thousands of lines of code and external tools. It is not clear whether their output can be fully trusted, despite the use of formal methods to generate schedules.

2.5 NeSTiNg

The analysis and validation of TSN scenarios may demand a high consumption of resources (e.g., cost, time, effort). The use of simulation tools is a viable solution to enable fast and easy validations and studies.

The use of a simulated environment gives to the user more freedom to create and validate more complex scenarios with less effort, preventing potential damage to the equipment that would be used in experimental scenarios.

The object-oriented event-based simulator OMNeT++ [22], is usually used to build network simulation models, having an IDE (Integrated Development Environment) based on the Eclipse platform. Each component is programmed in C++ and later assembled in bigger modules (e.g. devices and switches) using a high level language (NED – Network

Description). OMNeT++ also uses files to initialize the modules (INI – Initialization). When the modules in the high level language are joined to create a simulation scenario, the parameters of this modules must be informed in the respective initialization file.

The INET-Framework [23] is an open source set of modules and protocols for OMNeT++ capable of simulating wired, wireless and mobile networks. This framework is well known in the simulation community and used as base for countless simulation models.

Lastly, NeSTiNg [8] is a simulation model for OMNeT++ that uses the INET-Framework and enhances it adding components to enable TSN simulation. In addition to the files used by OMNeT++, NeSTiNg uses XML files to configure key components of the network, e.g. traffic generation, port scheduling (GCL – Gate Control Lists) and packet routing.

3 Related Work

As mentioned before, the purpose of TSN is to enable deterministic communication regarding latency and jitter. However, this objective can be splitted into three main contributions: i) The support of scheduled traffic; ii) convergence of different types of traffic in the same network; and iii) zero-loss reliability [24].

In this chapter we will highlight the state of the art regarding TAS schedule generation, TSN simulation and the studies regarding fault-tolerance. We also highlight relevant works in the Computer-Aided Verification filed, used to create the schedule validation.

3.1 TAS Schedule Generation

All contributions listed before are brought together in TSN most important component, the TAS schedule. This schedule contains all information necessary to control the forwarding gates and priority queues of the network switches. However, this schedule is computed offline, at design time and remains the same for the rest of the network execution, making it difficult to add or remove devices and flows from the network dynamically.

As mentioned on previous chapters, the creation of the TAS schedule is a difficult task, and can demand a large amount of time and resources. Synthesizing TAS schedules is a widely researched topic, where different research groups use different strategies.

Dynamic applications, such as the Industrial Internet of Things (Industry 4.0) can benefit from the scheduled traffic proposed by TSN [25]. However, these scenarios require an online flow scheduling, since new devices and flows can be added or removed constantly. This is called Plug-And-Produce configuration systems [26]. When this happens the network schedule must be recalculated, and this may take time.

The work presented in [27] propose an incremental schedule generator using Integer Linear Programming (ILP). When a static schedule is generated, modification can be made in an incremental fashion, reusing the previous schedule to generate a new one faster. ILP is a standard used for optimization problems applying solvers and mathematical formulations, the works proposed on [28] and [29] also use this technology to create TAS schedules.

A recent approach [30] uses a heuristic in combination with a “divide and conquer” approach for TAS scheduling which shows promising results. Such approach is achieved by dividing the scheduling problem into smaller, easier-to-solve subproblems and combining the results to produce the scheduling solution for the network. The approach is envisioned to be deployed on factory automation networks, which provides the necessary hierarchical structure to enable the fragmentation of the topology into subnetworks. Thus, providing the necessary grounds for an effective application of the divide and conquer methodology.

Another approach [31] generate TAS scheduling constraints using logic programming and synthesize a feasible schedule with these constraints. Similarly to TSNsched, this work also uses SMT solvers to generate schedules from the derived rules of the problem. However, as the other approaches using SMT solvers, TSNsched enables more types of properties to be specified. For example, [31] does not support explicit constraints regarding the jitter quality.

A key advantage of using SMT-based approaches is the existence of mature SMT solvers, such as Z3. This means that workflows using such tools can take advantage from improvements made directly in the solvers. However, TSNsched is open-source, whereas the tools produced by TTTech [2] [4] [5] [32] are not.

3.2 Simulating TSN

Generating the schedule for a TSN network is only part of the challenge. There are multiple ways to model this problem, as discussed previously in this chapter. Tools used to create network schedules usually cannot guarantee the effectiveness of the schedule by themselves. A way to prove the effectiveness of the schedules generated by these tools is by simulation.

The use of a simulated environment gives the user more freedom to create and validate more complex scenarios with less effort, preventing potential damage to the equipment that would be used in experimental scenarios.

CoRE4INET [33] is an extension to the INET Framework [23] for the OMNeT++ [22] simulation system. This simulation model provides applications and examples of protocols like AS6802, AVB, and recently TSN. But a weakness of this work is the network configuration not being user friendly, making it troublesome to create and run scenarios.

We can find a more recent and robust tool in NeSTiNg [8]. This tool is also an extension to the INET Framework, designed to support multiple TSN Standards, such as 802.1Qav, 802.1Qbv, and 802.1Qbu. The main difference in NeSTiNg is the method used to configure a scenario. NeSTiNg uses XML files to configure essential parts of the simulation, e.g., routing and port scheduling of the TSN Switches.

Other works such as [34] and [35] present tools capable to configure TSN scenarios in the NeSTiNg simulation model through a Graphical User Interface (GUI). This method makes the configuration more intuitive, but just like the other tools, the user needs to manually input the configuration.

The survey [24] mentions the use of simulation frameworks as an evaluation method. From the 49 simulation studies investigated, 23 used OMNET++, were 21 of these papers also used INET. Among the studies that used INET, 7 of them used NeSTiNg and only

3 used CoRE4INET. These numbers show how NeSTiNg has been accepted and used as a reliable opensource simulation framework for TSN.

3.3 Fault Tolerance Studies

Before going into fault-tolerance we need to define faults. According to [36], fault, failure, and error are used interchangeably, meaning either a hardware defect or a software/programming mistake (bug). The duration of faults can be classified into permanent, transient and intermittent. As the name suggests, permanent faults are the complete failure of a component, without recovery. The transient faults happen for an amount of time but can be recovered after it. An intermittent faults happens occasionally, never going away completely.

Among the traffic types that can occur on a Time-Sensitive Network, emergency traffic have the highest priority and may occur sporadically without prior notice. Given the nature of this traffic type, its priority is higher than the scheduled traffic, and since it occurs sporadically, it is not taken into consideration on the TAS schedule creation.

If an emergency message were created, there is a high chance this message will take the place of a scheduled message. So, this scheduled message would be postponed and take the place of the subsequent message, creating a domino effect and disrupting the entire schedule that were for that network.

The support for asynchronous emergency traffic in TSN was first introduced by [37]. In this work, the authors offer a solution in the form of a *Protection Band* working as a fail safe mechanism. In an extension of the previous work [38], the authors implement their solution on the Time-Aware Shaper. Even if both papers presented a solution to the asynchronous emergency traffic, both works have a similar flaw. The authors take in consideration single-upsets in the network schedule. Even though it rarely happens, if multiple faults were to occur at the same time, the solution proposed could not be applied.

The work presented in [39] shows the limitations of the packet preemption standard on TSN. Using simulations, they propose a novel frame preemption method able to improve the packet preemption in 90%.

As mentioned before, the network schedule is computed in advance, so if an error or change in the topology occur in run time, all the previous work could be discarded. Reference [40] uses Integer Linear Programming and a heuristic algorithm to achieve tun-time recovery of scheduled flows in case of network faults. Similarly, Reference [41] proposes a heuristic algorithm for online incremental rerouting and rescheduling of disrupted flows, assuming the paths and schedules of existing flows stay fixed.

The network may present many types of faults, that can result on latency increase or communication outage. The work [7] does an analysis of the redundancy management standard, describing the FRER mechanism and using simulations to demonstrate the effectiveness of this mechanism with and without fault-injection.

The replication technique used by TSN provides spatial redundancy. This increases the network reliability to deal with permanent faults. However, this approach may not present the same results for transient faults. The work [42] also goes over the redundancy management standard, but have a more theoretical approach, proposing a proactive retransmission of frames to tolerate transient faults.

Most of the studies found in the literature consider TSN fault-tolerance from a spacial and time redundancy perspective. Adapting the schedule and frame replication in run-time. As presented in the next chapters, we approach TSN shortcomings considering failures on network devices and infrastructure, analyzing the impact of these faults on the network and applying countermeasures during design time, when the TAS schedule is being generated.

3.4 Computer-Aided Verification

In general, the study of validation holds significant importance, as it serves to preemptively address or rectify any potential issues that may arise. There are several others studies in the area of computer-aided verification and checkers as shown in [44]. Most of them are not actually TSN specific, however they served to inspired our work.

The work [46] presented a tool that enables the developers to generate automatically reflective UML (Unified Modeling Language) State Machine controllers and the Runtime Safety Properties Checker (RSPC) which checks a component-based software system's safety properties defined at design phase. The checker detects when a system safety property is violated and starts a safe adaptation process to prevent the hazardous scenario. Thus demonstrating that the safety of the system is enhanced.

On the other hand, [45] presented a fully-automated static analysis tool capable of performing general bug finding using both pointer and taint analyses that are flow-sensitive, context-sensitive, and field sensitive in kernel drivers. A more close related work is presented by [47] which employed a probabilistic model checker using PRISM that work on network simulation by receiving the network as a kind of state machine and the queries about the probabilities sought in the form of logical formulas and then calculates the probabilities.

4 Fault-Injection Methodology for TSN Robustness Evaluation

The previous chapters present TSN standards, highlighting the fault-tolerant techniques. Even though TSN has a certain level of fault-tolerance, these techniques have limitations, and can only guarantee zero-loss reliability when the network is in prime conditions. In this chapter, we delve into the selected faults and enhancement made into the simulation model to enable fault injection.

4.1 Faults

The primary detrimental effects that faults can inflict on TSN is disrupting the deterministic communication. In other words, tempering with the network causing increase of latency and jitter. Latency is the time it takes a packet to reach its destination from the moment it is created, and jitter is the variation in this delay. Figure 8 shows four delays used in the latency calculation.

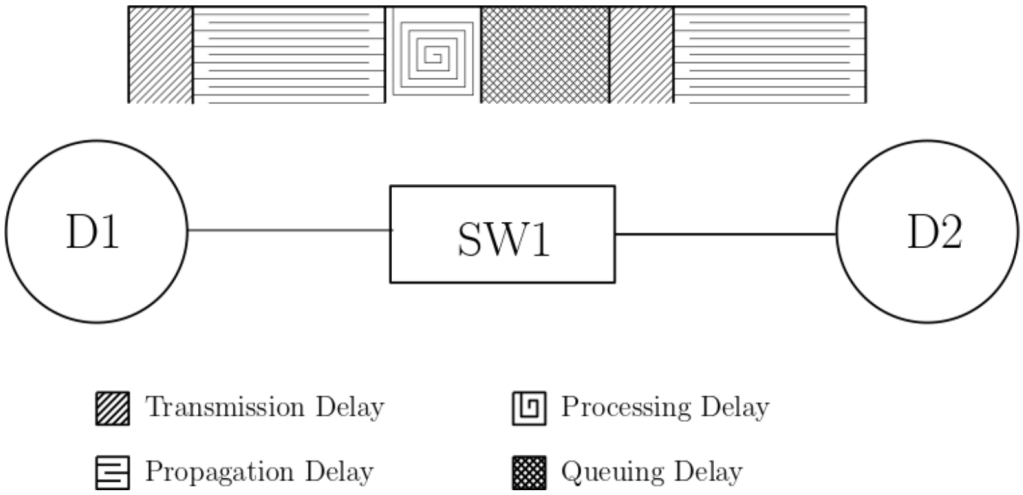

Figure 8: Delays used in the latency calculation.

The time required for all bits of the packet to be transported from the device to the transmission medium is referred to as Transmission Delay (D_{trans}). The Transmission Delay can be calculated by using (1), being P_{size} the packet size and T_{rate} the Data Transfer Rate.

$$(1)\ D_{trans} = \frac{P_{size}}{T_{rate}}$$

The Propagation Delay (D_{prop}) of a packet refers to the time used for the data to propagate on the transmission medium. The Processing Delay (D_{proc}) is the time where the switch processes the packet and sent it to the priority queue. The Queuing Delay

(D_{queue}) is given by the time the packet spends in the priority queue, waiting its turn to be sent to the next node. This delay depends on network traffic and schedule.

With the basic concepts and state of the art presented, we need to determine the faults we are going to use on this work. Taking inspiration on [7], the simulation model can now inject the following faults:

- **Switch Overload:** This is a transient fault. It happens when a switch takes too long to process the packet. When this happens, the processing delay of that switch will increase. If the packet takes too long to arrive at the priority queue (Figure 3), it may miss their send window allocated by the schedule.
- **Link Failure:** This is a permanent fault and this error happens at the switch port. When the network infrastructure present issues, the connection between devices may be severed. Without connection, all traffic that had to use that port is dropped.
- **Transmitter Failure:** This is an intermittent fault. This error happens when a transmitter send a packet with wrong header. The header of a TSN packet have valuable information (e.g. priority), so if a transmitter send a packet with a wrong header, the information may never reach the receiver or disturb the traffic flow of other priority.
- **Receiver Failure:** It is also an intermittent error. This fault happens when a receiving device fails to sort the received packets in the right order, dropping useful packets by mistake.

4.2 Simulation

To analyze the scenarios where the TAS schedule robustness is pushed to its limits, we made modifications to the simulation model NeSTiNg, creating new modules to emulate faults[1]. These new modules are able to inject faults into the network using a probability distribution with a seed provided by the user. So far we implemented the three following distribution into the simulation model:

- **Uniform Distribution:** This distribution implies that all outcomes within a given range have equal likelihood. The simplicity and effectiveness of this distribution are perfect for simulations and random sampling scenarios where each value has equal chance to occur.
- **Poisson Distribution:** The Poisson distribution is commonly used in situations where events occur randomly independent of time or space. Since this distribution

[1]https://github.com/piuMoreira/nesting-nestsched

is simple and future occurrences are not impacted by past events, it is a good choice for simulation of rare events.

- **Log-Normal Distribution:** This distribution is a little more complex when compared to the two previously mentioned. Using the Log-Normal distribution in simulations can help capture the underlying characteristics of the real-world data.

The faulty modules introduced in the simulation tool use the uniform distribution by default, but the user can select other distribution if they need. Different components can use different distributions, according to the user discretion. Also, even though we added three probability distributions to the simulation tool, new distributions can be added, turning this component scalable.

The flow of a packet inside the simulated switch is as follows. When a packet arrives at the switch port, emulated by the component named *eth*, it will go through the Ingress Traffic Conditioner (Figure 10). After this, it will pass by the *down* Message Dispatcher and *relayUnit* (Figure 9). The dispatcher is used to connect applications, automatically dispatching messages and packets among them. In the other hand, the relay unit is used to relay messages based on their destination MAC address.

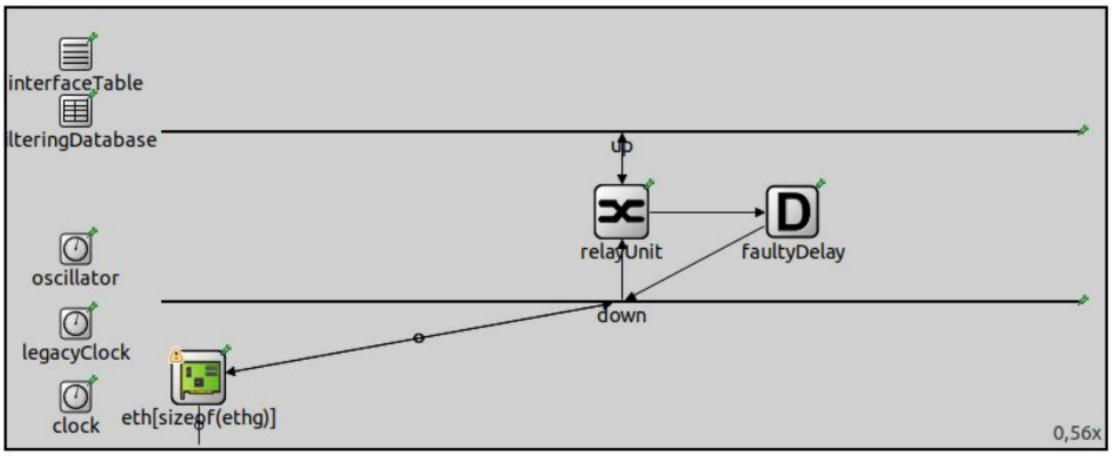

Figure 9: NeSTiNg simulated switch module.

After the packet is relayed, it will go through a *delayer*. This delayer component is used to emulate the processing delay present in real switches. The packet is then sent to the *queue* module inside the egress port. As we can see in Figure 11, inside the *queue* module each priority will have a *queue*, a *transmission selection algorithm* and a *transmission gate*.

As explained before, the queues are used to store the packets, when we use this together a transmission selection algorithm, we can give new characteristics to the queue. In the case of Figure 11, we are using a Strict Priority Algorithm, making the queue behave like a normal FIFO. The transmission gate state is controlled by the *gateController*, this

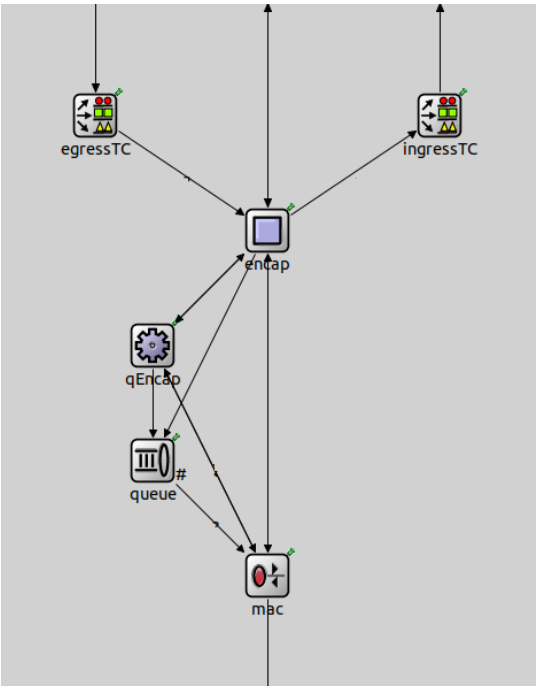

Figure 10: NeSTiNg simulated port module.

module contains the GCL (Gate Control List). Lastly, we have the *transmissionSelection* module, if multiple queues have an open transmission gate, this module guarantees the packet selected to transmission belongs to the queue with highest priority.

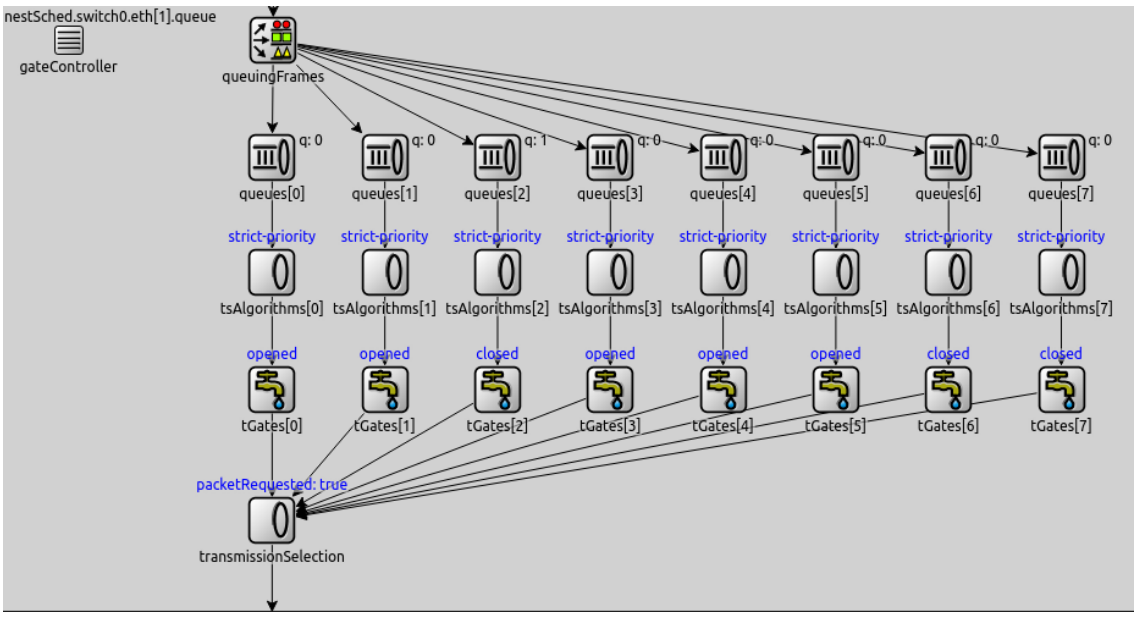

Figure 11: NeSTiNg simulated queues module.

To emulate the Switch Overload fault, we created a switch with a new delayer. In this new delayer, the static variable used to define the processing delay is now defined by the probability distribution and percentage selected by the user. The default distribution is the Uniform Distribution. However, other distributions can be found on our enhanced simulation model, leaving this choice for the user.

Similarly, the Link Failure fault was emulated by creating a new module for the switch ports. Whenever a packet arrives at the switch port, more specifically on the *mac* module, a function with a probability distribution is executed using the seed provided by the user. If the faulty component is triggered, all the traffic received by that port is dropped, losing the packet and skipping all the process described before.

The end devices in NeSTiNg use the same module to generate and receive traffic (Figure 12). To emulate the Transmitter Failure, the *trafGenSchedApp* module uses a probability distribution to check if it will send the packet with the right information. When receiving a packet, the module will check if the information on the header are correct and will also use a probability distribution to check if the packet will be dropped or not, emulating the Receiver Failure.

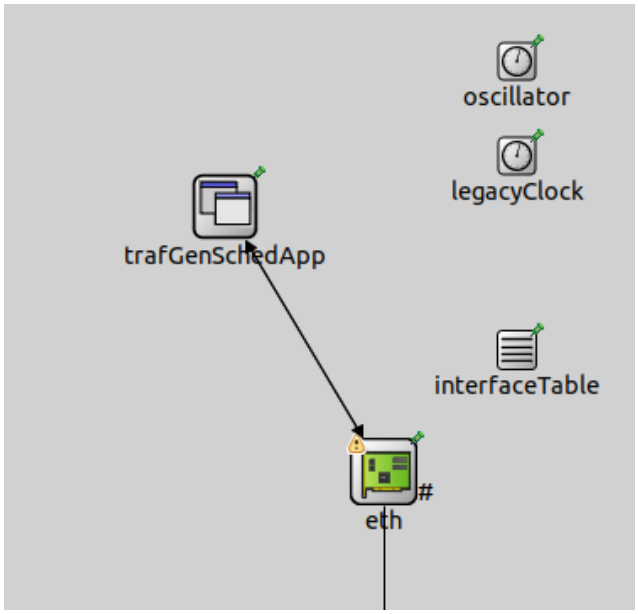

Figure 12: NeSTiNg simulated end-devices module.

4.3 Fault-Tolerance in Schedule Generation

TSNsched uses the network details as input to calculate the traffic schedule. This output is given as a JSON file, which is not supported by OMNET++. The process to translate the schedule to a simulation tool is tedious and error prone when done manually; thus we used a TSNsched plugin called Nest-Sched [43] to generate the simulation files required by NeSTiNg automatically. A visual representation of the tool-chain can be found in Figure 13.

As explained in Chapter 3, the Protection Band is a fail safe mechanism to ensure the traffic still meets the latency constrains established by the schedule. This mechanism is noting more than a time-slot were all the transmission gates are open. With this,

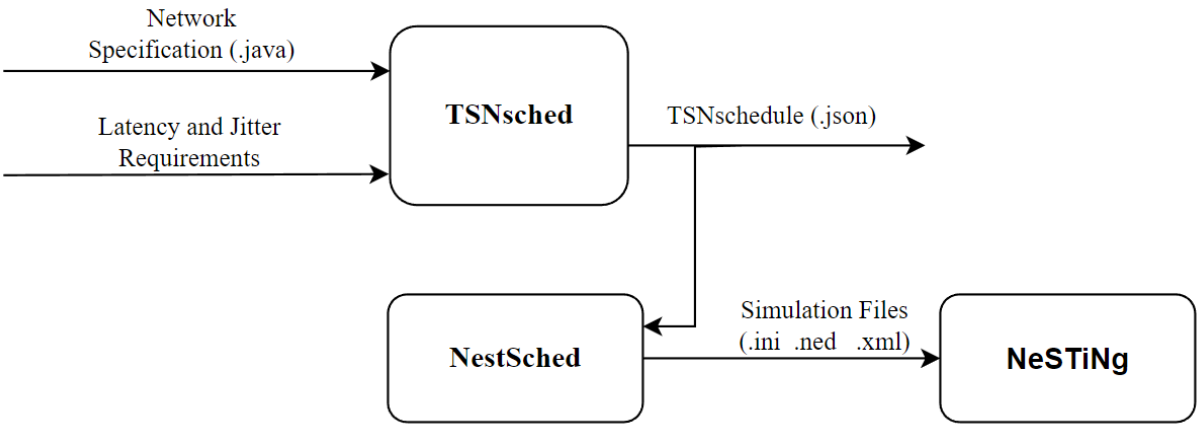

Figure 13: Tool-Chain.

packets that had to be interrupted or delayed have a chance be transmitted withing the same cycle, instead of waiting for the next cycle (Figure 14).

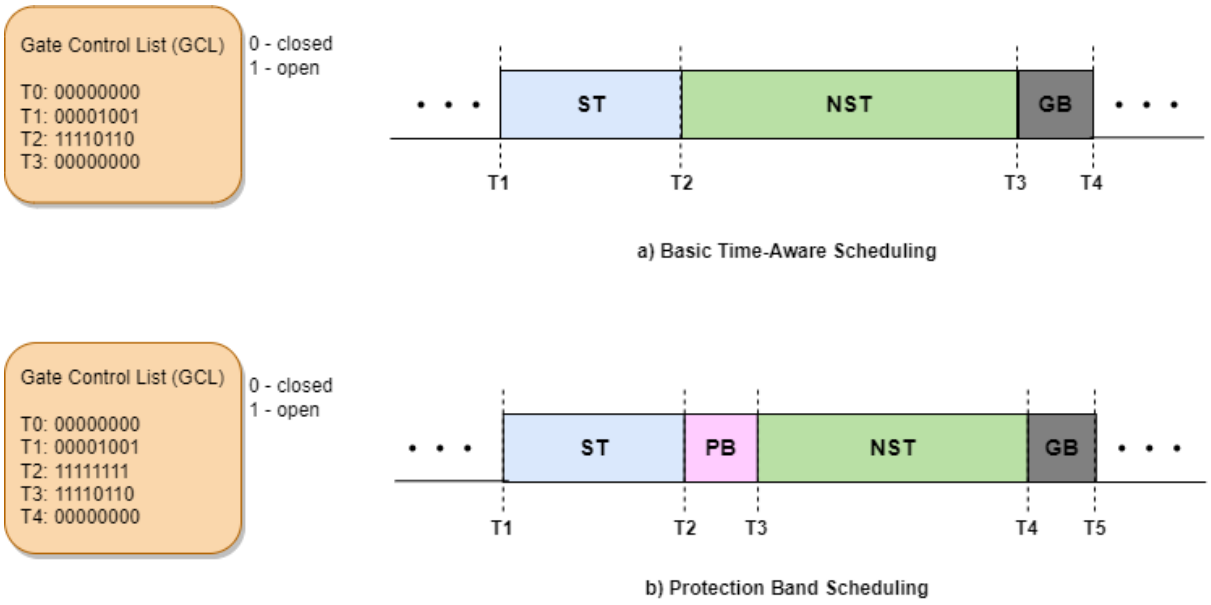

Figure 14: Basic TAS vs. Protection Band. Adapted from [37].

Inspired in works [37] and [38], we started to study the feasibility of the Protection Band on TSNsched. The way these works handle the Protection Band is enhancing the TAS with new rules to add the protection band dynamically at run-time when a emergency packet arrives. The cost to calculate and add a new time-slot and change the schedule throughout the network is high, so we try a new approach.

When TSNsched generates the schedule of a network, the cycle size can be determined using the Hypercycle and Mycrocycle approach (Chapter 2). Depending on the network details, the cycle can be too big for the scheduled traffic, resulting in a high allocation for non-scheduled traffic.

Figure 15 is an example of such scenario. The cycle size is determined as 400 μs but only 4 μs are used to scheduled traffic. Since the non-scheduled traffic have the lowest priority, we decided to take part of the non-scheduled traffic allocation to use for the protection band.

```
<port id="0">
  <schedule cycleTime="400.0us">
    <entry>
      <length>5.004us</length>
      <bitvector>11111110</bitvector>
    </entry>
    <entry>
      <length>4.0us</length>
      <bitvector>00000001</bitvector>
    </entry>
    <entry>
      <length>390.996us</length>
      <bitvector>11111110</bitvector>
    </entry>
  </schedule>
```

Figure 15: Port schedule generated by Nest-Sched.

Instead of dynamically adding the protection band on run time, we enhanced the Nest-Sched plugging to apply the protection band on design time. Once the schedule is generated by TSNsched and the simulation files are translated using Nest-Sched, the protection band can be added to the port schedule if the user set a variable on the TSNsched input. By adding the protection band on design time, we don't have the overhead of new schedule calculations, turning this approach simple and effective.

4.4 Schedule Verification

TSNsched generates two files when executed, the network schedule in the form of a JSON (JavaScript Object Notation) file and a log file. The amount of validations needed can be overwhelming to a normal user, so we considered an approach to ease management and issue tracking. The approach used is to categorize the validations made by the Checker into two major groups: Semantic and Syntactic. These categories are presented in more details in Sections 4.4.1 and 4.4.2.

4.4.1 Syntactic

As the name suggests, this category validates the syntactic aspect of the files, for instance, misspellings and incorrect data types. This checker receives two files from the user and the first step is to ensure all data meet the requirements, such as data and file types to proceed the validations as needed for the semantic validation. Syntactic errors

are easier to be noticed as they raise attention as soon as spotted; for instance, some letters in the place of a number.

1. **Data Type:** As mentioned earlier, a letter in the place of a number, and other situations where we do not get what is expected, is a syntactic error. The checker verifies the data type of information contained on the file to ensure they are correct.

2. **File Type:** Just like the one before, this validation now aims for the files types. This checker receives a JSON and a log files that can have the .log format extension or a simple text (.txt) extension. So we make a quick verification to ensure they correspond to these extensions.

4.4.2 Semantic

The semantic validations mean the correctness of the data into a deeper analysis. The semantic differs from the syntax at adopting a more technical view of the situation, so it may not be so clear to the user that does not comprehend the behavior of the TSNSched, some errors may even pass unnoticed by experienced users as the topology becomes too large. To help the user, this semantic validations target logical question, such as, whether the transmission of a packet respects the transmission window of its port.

1. **Switches:** A switch stores the data related to the cycle duration, the priorities assigned to flows, and the TSN time slots allocated for each priority at each port.

    - **Ports:** The ports contain important information, such as, packet transmission start, its cycle and transmission duration (also known as transmission window).

2. **Flow:** This is a topic which validates the flow data. The JSON file contains the average latency, jitter, data time of the packets and hops to end devices. The complete details of each flow fragment can be found on the log file, which provides the flowfragment information needed for further verifications. The flow then can be divided into a couple more specific subtopics, they are: Paths and Packet Times.

    - **Paths:** The flows contain data about the hops from the origin device to the end device. It describes each hop from the origin device to the end device by presenting the current and destination nodes, and the priority of the hop. Considering that, it might happen that a node may be missing or the scheduling inserts a node in the wrong place the checker validates the path to the end devices ensuring it follows as pretended.
    - **Packet Times:** A flow is broken into fragments, each fragment containing data of the hops. The JSON file includes the time when it left the previous

node (*departure time*), the time it arrived at the current node (*arrival time*), and the time when it was sent to the next node (*scheduled time*). The log file presents more details of the flow fragment, such as the current hop (the origin and destination nodes) and the priority of the fragment. This information is the core of TSNsched, fitting them on a schedule without inconsistencies is vital for the deterministic network desired, nonetheless it is important to verify them to ensure correctness and reliability of the schedule.

4.4.3 Verification Criteria

The Checker works based on six criteria defined for evaluation:

- **Type checking value:** Checks for any negative number on the output because it is an impossible value of time when considering the real world scenario. Every timestamp is relative to at least its own start, therefore the minimum value is 0. Both log and JSON files brings us the data needed to verify.
- **Well formed hops:** On the output there are three variables of time on the flow fragments, they are: the arrival time, departure time and scheduled time. Those three are chained together since the arrival time represents when the packet arrived at the current node, the departure time represents when the packet left the previous node and the scheduled time represents when the packet is to be sent to the next node. The tool checks if the nodes are well scheduled by comparing this information of a node with its previous, more precisely, there is a need to ensure that the departure time of a node is equal to the scheduled time of the previous node as seen in Figure 16 otherwise there is a scheduling problem.
- **Consistent path nodes:** It might happen that TSNsched inserts a node on the path to a certain device that is not supposed to exist. Therefore, to avoid packets from following a wrong route, we need to check whether the paths are consistent with its schedule. The Checker validates this by comparing the path to each device with the flow fragments as depicted in Figure 17.
- **Transmission windows consistency - Cycle:** There cannot be two or more packets being transmitted at the same time on the same port, and the transmission cannot overpass its cycle duration. For instance, if a packet is to be sent somewhere between 0 and 500 μs, it cannot be sent after this interval. As we can see in Figure 18, each switch have an array of ports, each containing an array of slots. Inside *prioritySlotsData* there is the *slotsData* for each transmission, it shows us the start and the duration of the transmission. The purpose here is to validate each *slotsData*

to ensure their start is not the same and also if they do not finish their transmission after the cycle is over.

- **Transmission windows consistency - Priority Window:** Each packet has a priority assigned to it, and so it must be transmitted at its correspondent priority window. This checker parses the JSON file looking for the flows and switches data to get the priority and node of each hop. By parsing the flow data, it takes each hop and save its priority, then compares whether there is a port on a specific switch transmitting a packet with the saved priority.

- **Transmission windows consistency - Sent Order:** When there are multiples packets on a port with the same priority, we must ensure they are being sent in the order of arrival just like a queue, where the first that comes in is the first that goes out (FIFO). This checker validates it by comparing the start of transmission of each slot data on a port.


```
{
   "flow0Fragment1": [
    {
      "packet0ArrivalTime": 8.576,
      "packetNumber": 0,
      "packet0DepartureTime": 0.576,
      "packet0ScheduledTime": 9.152
    }
  ]
},
{
 "flow0Fragment3": [
    {
        "packet0ArrivalTime": 17.152,
        "packetNumber": 0,
        "packet0DepartureTime": 9.152,
        "packet0ScheduledTime": 17.728
        }
    ]
},
```


Figure 16: Packet arrival, departure and scheduled times.

```
 Path to dev5:
dev38,
switch7(flow1Fragment1),
switch0(flow1Fragment2),
switch1(flow1Fragment3),
dev5,

Fragment name: flow1Fragment1
        Fragment node: switch7
        Fragment next hop: switch0
Fragment name: flow1Fragment2
        Fragment node: switch0
        Fragment next hop: switch1
Fragment name: flow1Fragment3
        Fragment node: switch1
        Fragment next hop: dev5
```

Figure 17: Path and flow fragments from dev38 to dev5.


```
"prioritySlotsData": [
    {
      "slotsData": [
        {
          "slotDuration": 0.576,
          "slotStart": 9.115
        }
      ],
      "priority": 7
    },
    {
      "slotsData": [
        {
          "slotDuration": 0.577,
          "slotStart": 8.843
        }
      ],
      "priority": 7
    }
  ]
```


Figure 18: Slot data information.

5 Experimental Evaluation

In this chapter, we describe the simulation setup, the network specification and expected latency. Then, we inject faults on the network and evaluate the impact they had on the overall latency.

5.1 Simulation Setup

We use the tool TSNsched to generate the schedule for the network depicted in Figure 19. The network have a constant data transfer rate of 1 Gbit/s, 10 end-devices, 4 switches and 10 traffic flows with different priorities, packet length and periodicity. The

details about the network can be found in Table 2. The Start and End columns represent the transmitter and receiver devices. The Priority column specifies the priority of that flow. Finally, the columns Size and Interval inform the length that flow packages and the periodicity which they are sent.

Table 2: Traffic flow details.

	Start	End	Priority	Size	Interval
flow1	dev0	dev5	0	500 B	400 μs
flow2	dev1	dev3	2	300 B	800 μs
flow3	dev3	dev1	1	300 B	800 μs
flow4	dev2	dev0	2	300 B	500 μs
flow5	dev4	dev6	2	400 B	800 μs
flow6	dev6	dev4	6	600 B	800 μs
flow7	dev5	dev0	6	700 B	500 μs
flow8	dev7	dev9	0	500 B	800 μs
flow9	dev9	dev7	0	300 B	800 μs
flow10	dev8	dev0	7	800 B	500 μs

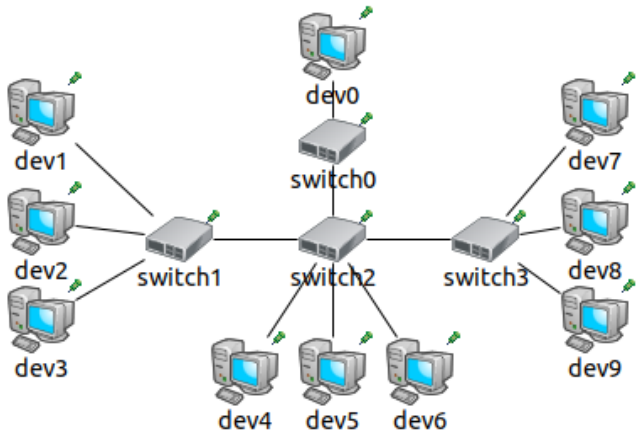

Figure 19: Network topology used for simulations.

5.2 Fault Injection

With the required files loaded into the simulation tool, we are able to execute and validate the schedule generated by TSNsched. The graph depicted in Figure 20 represents the network latency working in perfect conditions. As one can see, all the flows remain stable, meeting the specified constraints.

Having the schedule and the necessary configuration files for the simulation, we can inject the faults. As mentioned before, new components were created to enhance NeSTiNg simulation model, each new component uses a uniform probability distribution

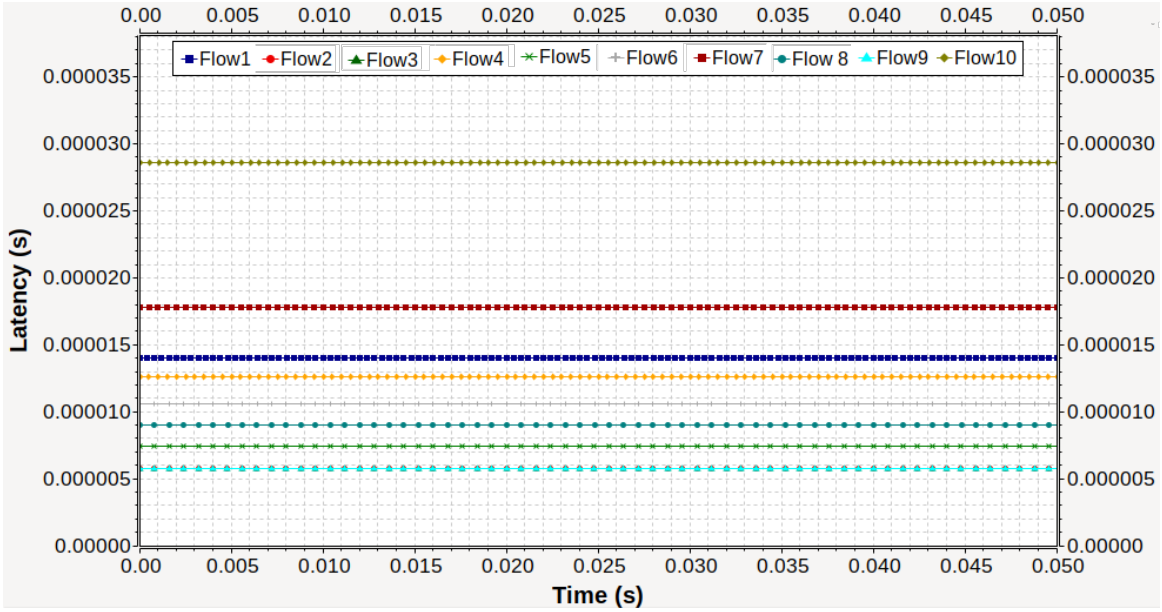

Figure 20: Network expected latency.

by default to determine if it will behave normally or trigger a fault. Thus, the user needs to specify the fault probability of said components.

OMNET++ simulation tool uses initialization files to specify device parameters. Using this, we can easily specify the fault probability of the simulated scenario. An example of this configuration can be seen in Figure 21.

```
**.switch0.faultyDelay.delayRng = 400        # microseconds
**.switch1.faultyDelay.delayRng = 700        # microseconds
**.switch2.faultyDelay.delayRng = 300        # microseconds
**.switch3.faultyDelay.delayRng = 900        # microseconds

**.switch*.faultyDelay.faultProb = 1         # %
**.switch2.eth[1].mac.dropProbability = 5    # %
```

Figure 21: NeSTiNg fault configuration.

5.2.1 Switch Overload

Using the new switch model, we are able to simulate a scenario were the network switches experience an overload, taking more time to process packets, sending them to the priority queue with delay.

Each switch of the network can have different probabilities assigned to them. When the switch fault is triggered, the received packet will take longer to arrive at the priority queue, and this delay range can vary, according to a value given by the user in the initialization file.

For the first fault scenario, we set the delay probability of all switches to 1%, and the delay range of these switches vary from 300 μs to 700 μs. These delay ranges were

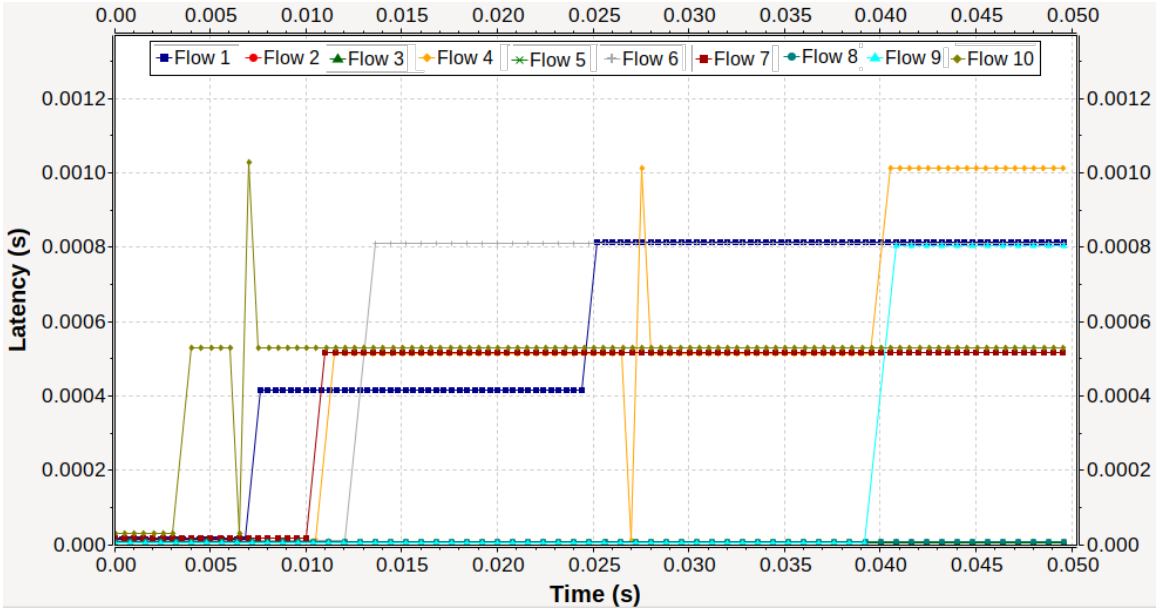

Figure 22: Network latency with switch overload fault.

selected taking in consideration the flows periodicity. The resulting latency can be seen in Figure 22.

The harmful strictness of the TSN scheduling mechanism is shown in this scenario. Applying a fault probability of 1% to the switches was enough to disrupt the schedule, increasing the overall latency and jitter. However, as we can see in Figure 22 and Table 3, not all dataflows were affected. Some dataflows remained with the expected latency and jitter due to the random aspect of the fault, in other words, the fault was not triggered for these flows.

The package size, network bandwidth and switch processing delay are used as input to create the network schedule, and these parameters are seen as static values. Thus, if there is any variation to these parameters at run time, the TAS will not be able to work around it since the GCLs are already loaded and strictly followed. Now, lets say that the network stability has worsened and all the switches are operating under critical conditions (70% of switch overflow). Instead of a cascading delay, the affected flows will present a worse variation, as depicted in Figure 23.

5.2.2 Link Failure

To emulate the link failure, the fault injector uses the new switch port model, which have a fault probability associated to it. When the fault is triggered, that port connection is severed, dropping any packet that is suppose to use it.

For this scenario, we injected the link failure fault on one port of the central switch, more specifically the port connecting Switch2 to Switch0. As we can see in Figure 24, after some time, the fault is triggered and the flows directed to dev0 are interrupted.

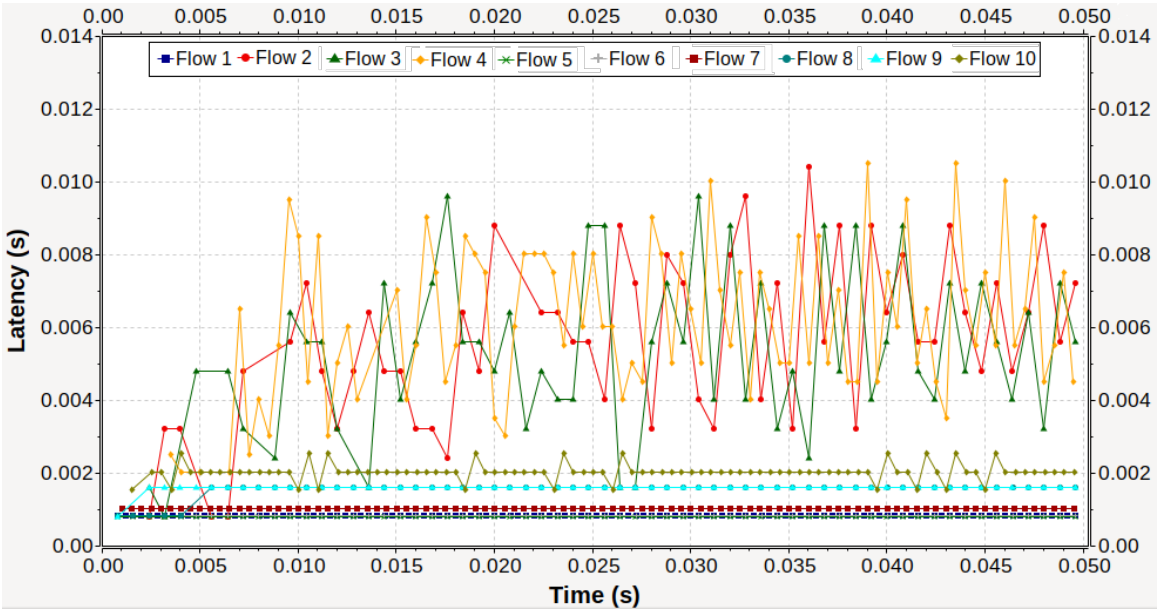

Figure 23: Network latency for critical case of switch overload.

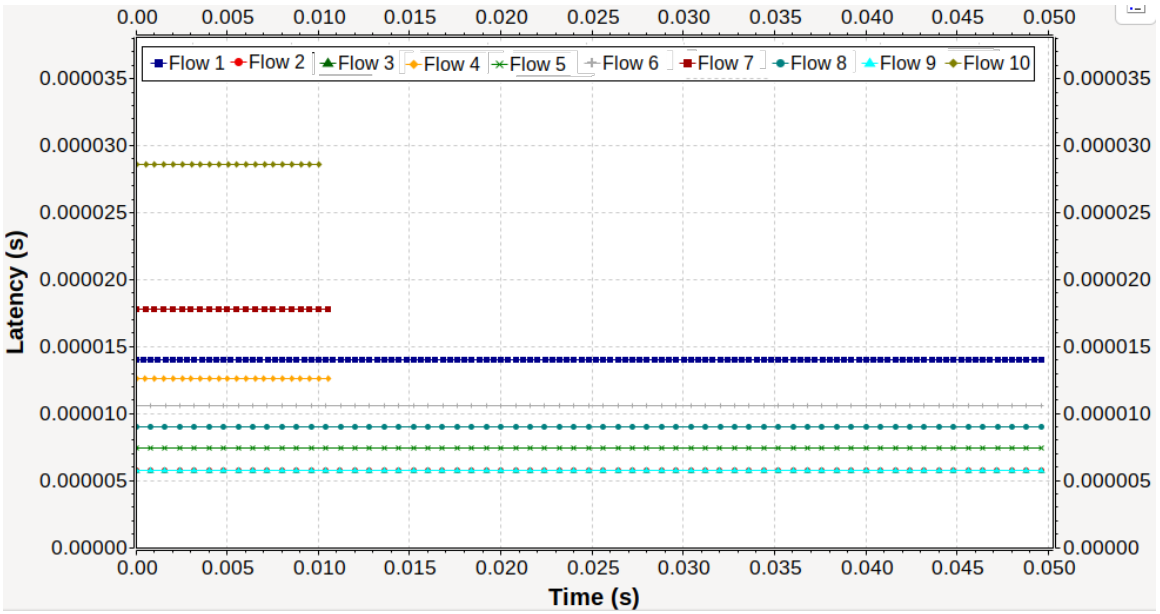

Figure 24: Network latency with link failure.

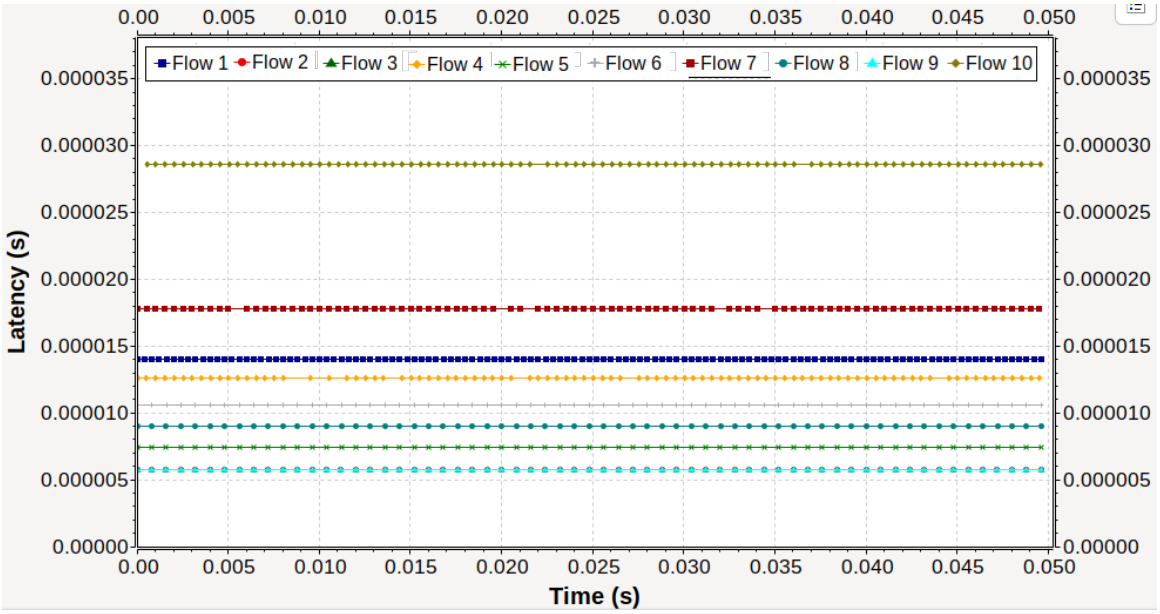

Figure 25: Network latency with transmitter/receiver failure.

This network topology (Figure 19) was used to show that TSN faults can be introduced even at the early stages of modeling. By default, TSN have a redundancy standard (IEEE 802.1CB) as a fault-tolerant measure. But this standard cannot be used to its full potential if the network topology does not offer multiple paths which a packet can be sent. If the topology was designed with multiple paths, the FRER mechanism could send packets trough different paths, allowing packet to be transmitted.

5.2.3 Transmitter/Receiver Failure:

The emulation of these faults are similar. The NeSTiNg simulation model uses the same component to transmit and receive traffic, so we injected the fault in this component.

Figure 25 depicts the network latency after injecting this fault. As we can see, none of the flows had latency variation, but the gaps on flows 4, 7 and 10 represent failure to transmit or receive a packet. This means the end device will not receive all the packets, resulting in information loss.

5.3 Latency Results

The latency results of all scenarios are summarized in Table 3. It shows the network expected performance and the results after the fault injection. As mentioned previously, the only scenario where the latency was impacted is the Switch Overload. This is caused by the unexpected delay added throughout the path.

In the scenarios where there was packet loss, the latency remains stable, this happens because the simulation tool only uses the received packets in the latency calculation;

Table 3: Latency results.

	Scenario							
	Expected Result		Switch Overload		Link Failure		Transmitter/Receiver Failure	
	Latency (μs)	Jitter (μs)	Latency (μs)	Jitter (μs)	Latency (μs)	Jitter (μs)	Latency (μs)	Jitter (μs)
Flow1	14.008	0	557.097	290.302	14.008	0	14.008	0
Flow2	5.792	0	5.792	0	5.792	0	5.792	0
Flow3	5.792	0	5.792	0	5.792	0	5.792	0
Flow4	12.592	0	497.285	332.547	12.592	0	12.592	0
Flow5	7.392	0	7.392	0	7.392	0	7.392	0
Flow6	10.592	0	604.14	352.913	10.592	0	10.592	0
Flow7	17.804	0	411.743	205.445	17.804	0	17.804	0
Flow8	8.992	0	8.992	0	8.992	0	8.992	0
Flow9	5.792	0	160.63	318.643	5.792	0	5.792	0
Flow10	28.601	0	493.248	147.297	28.601	0	28.601	0

so if a package is lost along the way it will not be counted. If we take in consideration the lost packets, the latency experienced is *Infinite*, since the packet will never reach the destination.

Even though the other scenarios do not present latency variation, the traffic flows are still being affected since the packets are not arriving at their destinations and information is lost.

5.4 Protection Band

Using the aforementioned technique, we can decrease the impact caused by the switch overload. If a packet arrives late at the priority queue and misses the transmission window, they can still be sent using the protection band. In this experiment we used the same topology presented on the beginning of this chapter, and a protection band of 10 μs.

In the example shown in Figure 26 (a), the send window was scheduled with the exact amount of time necessary to send Pkt1, Pkt2 and Pkt3. However, due to a delay on the switch, these packages arrive in the queue later than expected, causing Pkt3 to be delayed to the next cycle. With the introduction of the Protection Band, the packet can be sent in the same cycle, significantly reducing the delay as depicted in Figure 26 (b).

As we can see in Figure 27, the impact caused by the switch overload is much smaller than seen in Figure 22. Table 4 gives the detailed results of the protection band, comparing with the previous results. Even though there is still latency variation, the recovery time is much smaller, proving an increase on the schedule robustness and reliability.

After seeing the results of the protection band, we can confidently say the scheduled traffic latency had a big improvement. However, to create the protection band we take part of the non-scheduled bandwidth. This raises the question: How is the non-scheduled traffic affected by this technique?

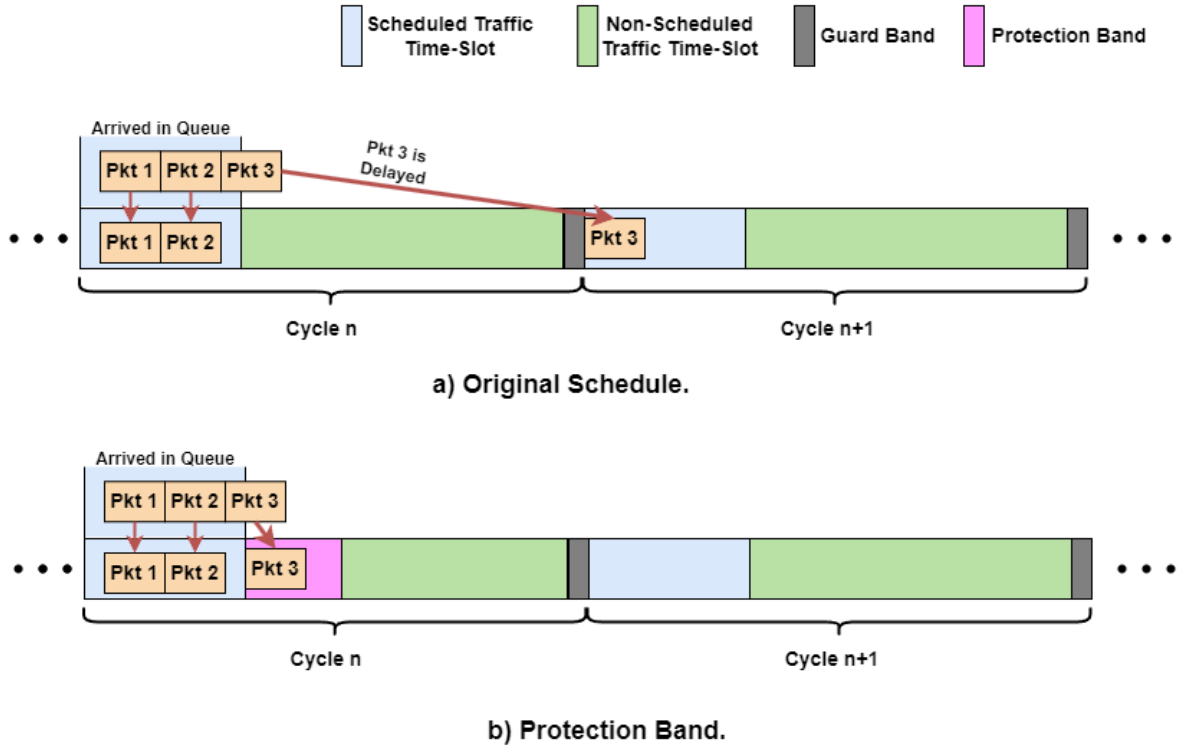

Figure 26: Protection Band comparison.

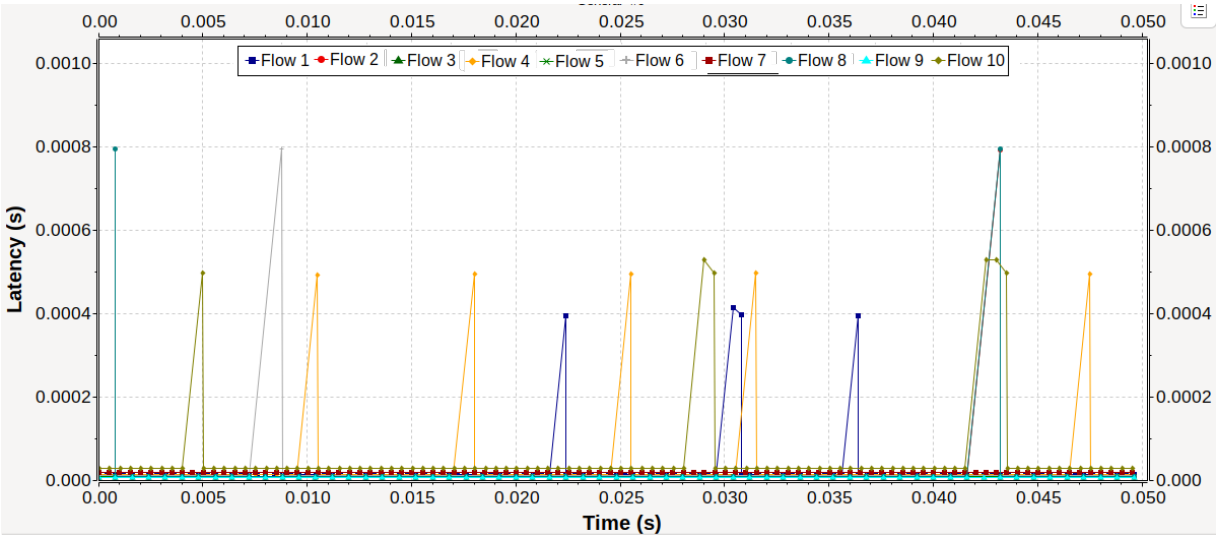

Figure 27: Network latency applying Protection Band.

Using the Non-Critical modules given by Nest-Sched, we are able to emulate non-scheduled traffic. In Figure 28 we can see the addition of six new devices that will generate the non-scheduled traffic. The non-scheduled traffic does not use static variables to define packet interval and size. Instead, the value set by the user is used a seed on a Poisson Number Generator, randomly assigning values to the non-scheduled traffic throughout the simulation execution. Table 5 has the details of the non-critical flows and seeds used.

As we can see in Table 6, the addition of the protection band had an impact in

Table 4: Protection Band results.

	Scenario					
	Expected		Switch Overflow		Protection Band	
	Latency (μs)	Jitter (μs)	Latency (μs)	Jitter (μs)	Latency (μs)	Jitter (μs)
Flow1	14.008	0	557.097	290.302	26.359	68.224
Flow2	5.792	0	5.792	0	18.227	99.103
Flow3	5.792	0	5.792	0	5.792	0
Flow4	12.592	0	497.285	332.547	36.702	105.62
Flow5	7.392	0	7.392	0	7.392	0
Flow6	10.592	0	604.14	352.913	23.039	98.8
Flow7	17.804	0	411.743	205.445	17.804	0
Flow8	8.992	0	8.992	0	33.913	138.735
Flow9	5.792	0	160.63	318.643	5.792	0
Flow10	28.601	0	493.248	147.297	57.635	115.565

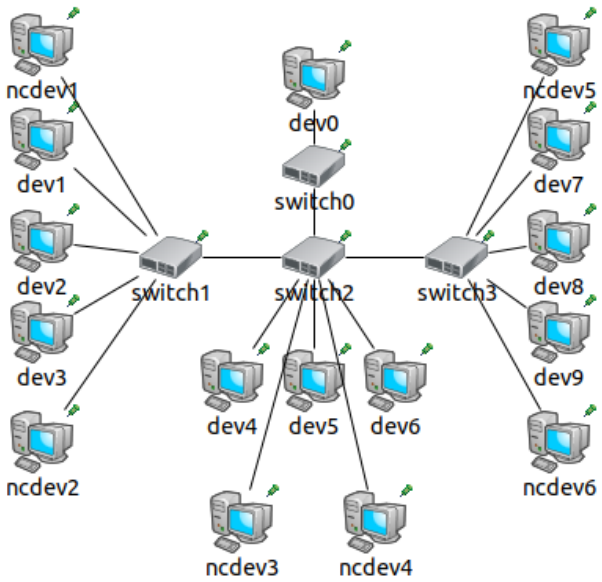

Figure 28: Updated network topology used for simulations.

Table 5: Non-critical traffic flow details.

	Start	End	Priority	Size	Interval
NC-flow1	NCdev1	dev0	5	700 B	19 μs
NC-flow2	NCdev2	dev2	5	700 B	19 μs
NC-flow3	NCdev3	dev0	5	700 B	19 μs
NC-flow4	NCdev4	dev5	5	700 B	19 μs
NC-flow5	NCdev5	dev0	5	700 B	19 μs
NC-flow6	NCdev6	dev8	5	700 B	19 μs

the non-critical traffic. But given the fact this traffic have the lowest priority among the traffic types, this increase is acceptable.

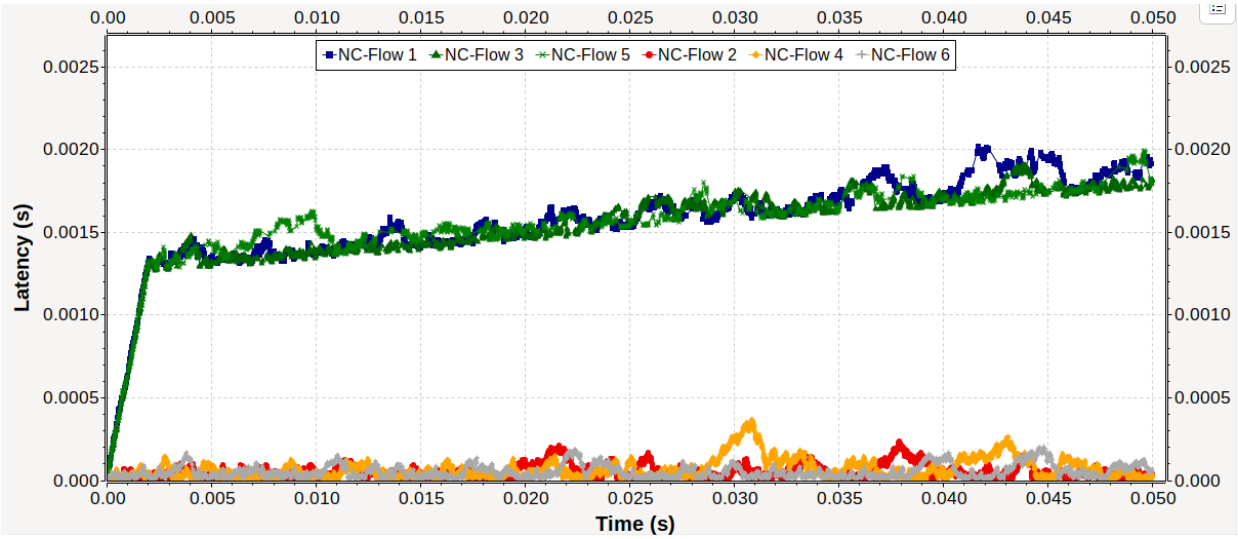

Figure 29: Non-critical traffic latency without Protection Band.

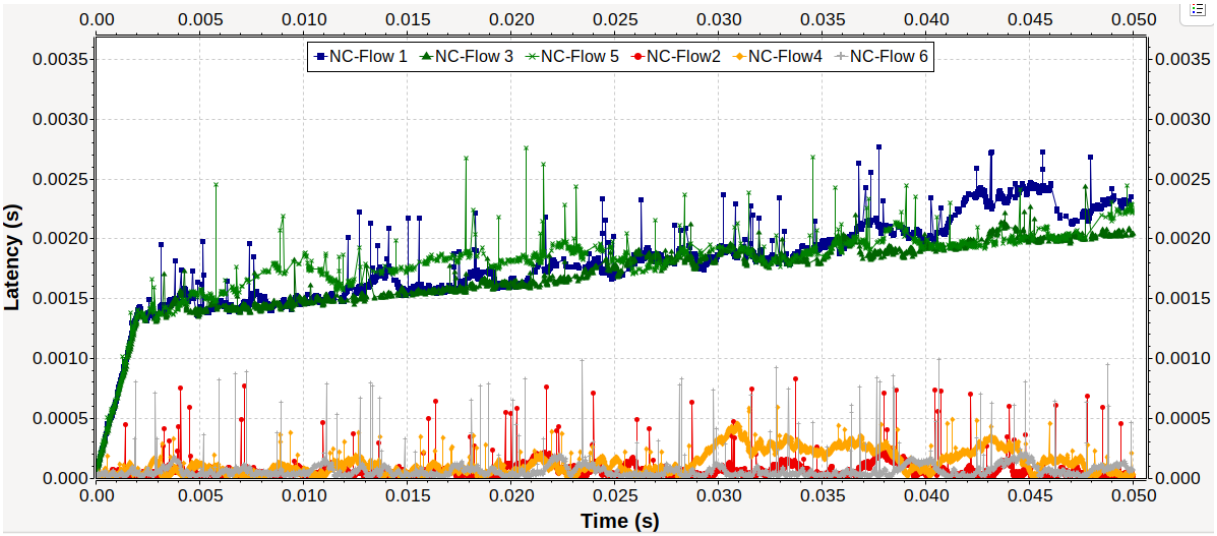

Figure 30: Non-critical traffic latency with Protection Band.

Table 6: Non-critical traffic results.

	Scenario			
	Without Protectin Band		With Protection Band	
	Latency (μs)	Jitter (μs)	Latency (μs)	Jitter (μs)
NC-Flow1	1577.441	255	1777.86	368.057
NC-Flow2	56.190	41.5	61.654	57.094
NC-Flow3	1559.193	242.209	1706.384	297.685
NC-Flow4	71.984	60.349	1275.572	96.723
NC-Flow5	1535.527	265.304	1791.672	328.891
NC-Flow6	56.849	39.412	63.342	67.061

5.5 Schedule Modifications

To the inexperienced user, the delay issue presented in Section 5.2.1 could be fixed by simply increasing the transmission window of the scheduled traffic using the

non-scheduled allocated bandwidth. However, this thoughtless approach have two main issues: (i) the priority distribution and transmission window balance are broken since the original schedule is grossly modified; (ii) with a transmission window imbalance, the waiting time of packages may increase instead of reduce.

Schedule generator tools such as TSNsched, aim to create a schedule with the lowest possible bandwidth waste. This means, the transmission windows of the scheduled traffic have the exact amount of time necessary to send the package, as depicted in Figure 31 (a).

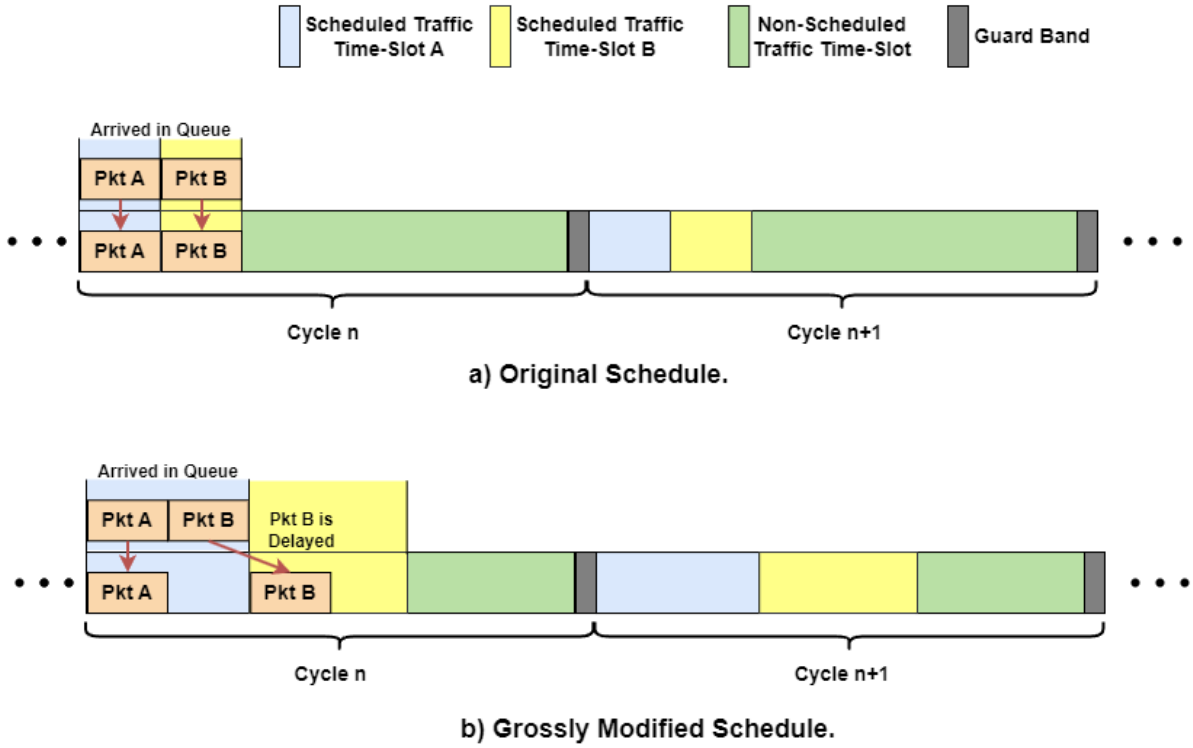

Figure 31: Grossly modified schedule comparison.

This scheduling choice, only takes in consideration the propagation, transmission and processing delay. If the transmission windows were to change in a crude manner, packets that were suppose to be sent with no delay in the original schedule, may now experience delay, as shown in Figure 31 (b).

Techniques such as Protection Band, are developed after careful evaluation. This technique works because there is no change to the transmission windows of the scheduled traffic. And all queues are open in the protection band send window (Figure 14), so the original schedule is preserved and if there is no critical traffic to be sent, the non-critical traffic and priority distribution can be respected.

Using the simulation tool, we were able to emulate the network behavior in cases were the schedule is grossly modified. Using the topology from Figure 19, we ran simulations were we increase the send window of scheduled traffic by 5% and 10% of the

non-scheduled send window, using an approach similar to Figure 31 (b). The latency graphs of these scenarios can be seen in Figures 32 and 33.

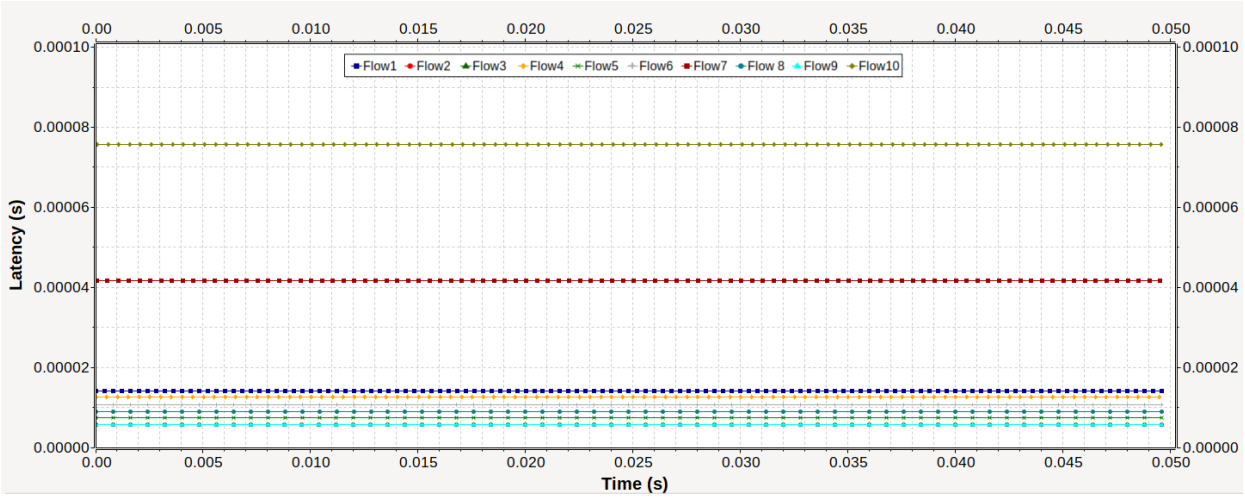

Figure 32: Grossly modified schedule - 5% increase.

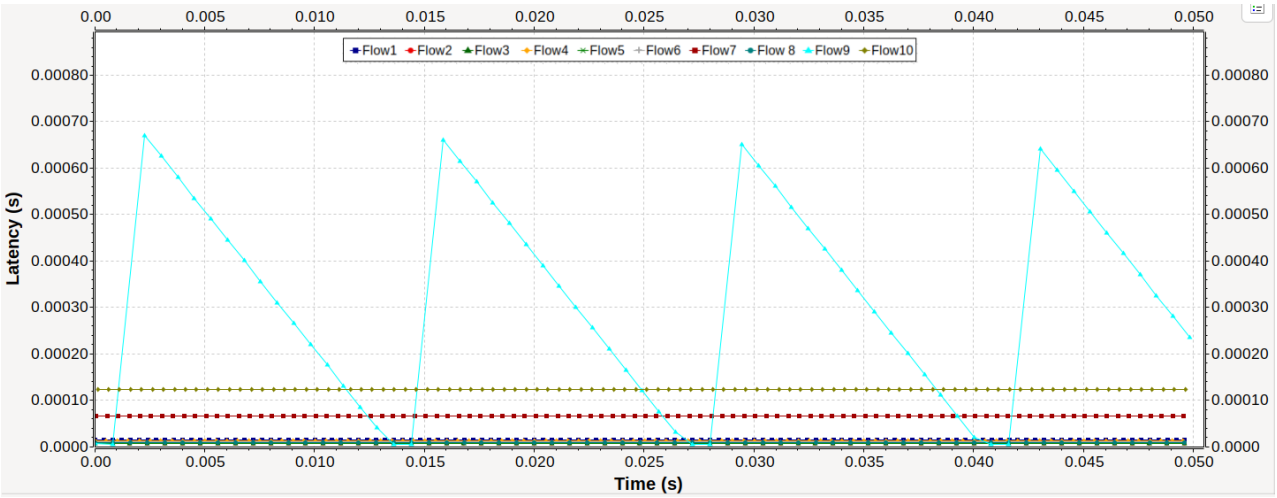

Figure 33: Grossly modified schedule - 10% increase.

As we can see in Figure 32, increasing the scheduled window by 5%, keeps the latency of all data flows stable. However, we can see a considerable latency increase in some of the flows. In the other hand, when we increase the send window by 10%, latency variation starts to appear, like depicted in Figure 33. The results can be found in Table 7.

With these results, we can see the impact of a crude technique in the scheduled traffic. The initial goal of this technique was to fix the delay issue presented in Section 5.2.1. So, this raises the question: 'What happens when we introduce the Switch Overload fault in this scenario?'. To answer this question, we ran the scenario again, but this turn we enabled the faulty delayer, emulating the Switch Overload. The latency results can be found on Figures 34 and 35.

Table 7: Grossly modified schedule results.

	Scenario					
	Expected		5% Window Increase		10% Window Increase	
	Latency (μ s)	Jitter (μs)	Latency (μ s)	Jitter (μs)	Latency (μ s)	Jitter (μs)
Flow1	14.008	0	14.008	0	14.008	0
Flow2	5.792	0	5.792	0	5.792	0
Flow3	5.792	0	5.792	0	5.792	0
Flow4	12.592	0	12.592	0	12.592	0
Flow5	7.392	0	7.392	0	7.392	0
Flow6	10.592	0	10.592	0	10.592	0
Flow7	17.804	0	41.743	0	65.683	0
Flow8	8.992	0	8.992	0	8.992	0
Flow9	5.792	0	5.792	0	317.189	213.821
Flow10	28.601	0	75.74	0	123.356	0

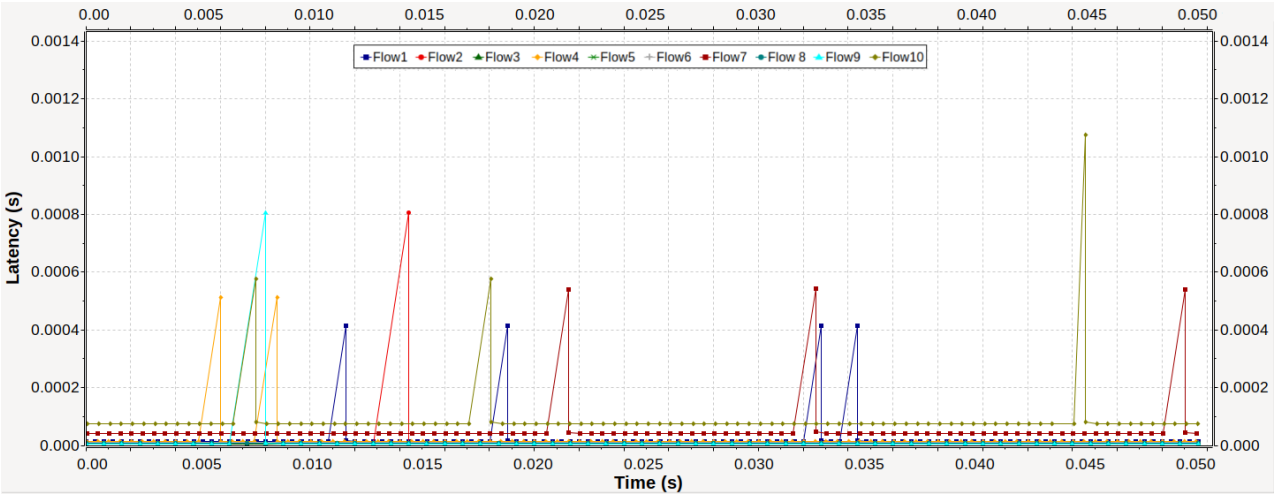

Figure 34: Grossly modified schedule - 5% increase with switch overload.

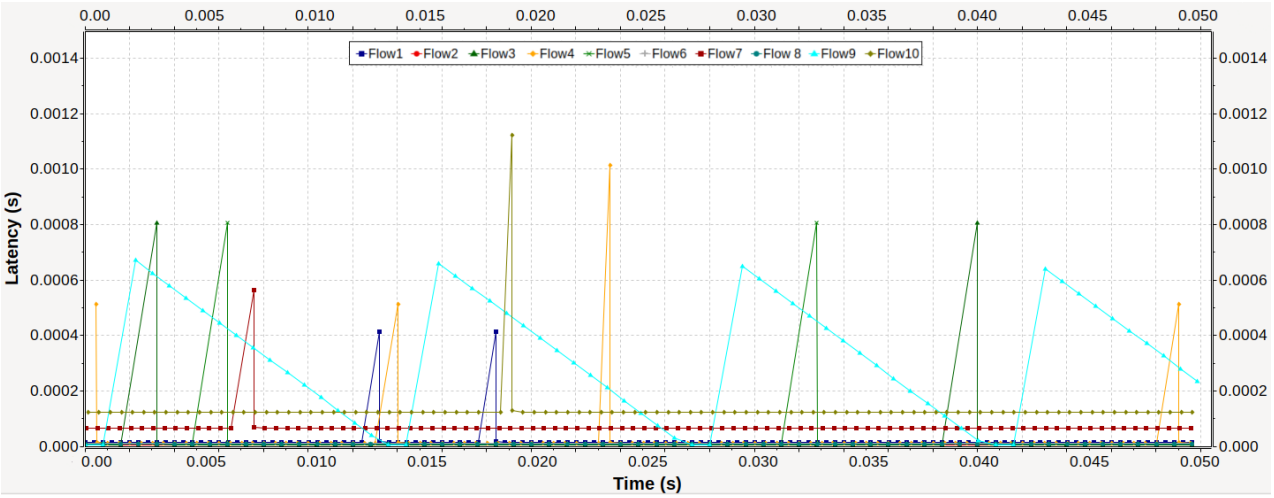

Figure 35: Grossly modified schedule - 10% increase with switch overload.

As we can see in the graphs, the recovery time of the latency in the fault scenario is impressive. As we can see in Table 8, the latency results of the window increase are

comparable with the results found with the Protection Band in some cases. Faults may happen at any time during the network execution, but the probability of a fault occurring is low. So, even though the latency recovery is fast, the general latency increase observed in the network (Table 7) makes this technique hazardous.

Table 8: Comparison of Protection Band and window modification results.

	Scenario					
	Protection Band		5% Window Increase With Switch Overflow		10% Window Increase With Switch Overflow	
	Latency (μ s)	Jitter (μs)	Latency (μ s)	Jitter (μs)	Latency (μ s)	Jitter (μs)
Flow1	26.359	68.224	26.935	70.663	20.472	50.386
Flow2	18.227	99.103	18.528	100.786	5.792	0
Flow3	5.792	0	5.792	0	31.265	141.371
Flow4	36.702	105.62	22.64	70.346	37.687	130.54
Flow5	7.392	0	7.392	0	32.89	141.367
Flow6	23.039	98.8	10.592	0	10.592	0
Flow7	17.804	0	56.809	85.413	70.680	49.705
Flow8	33.913	138.735	8.992	0	8.992	0
Flow9	5.792	0	18.528	100.786	317.189	213.821
Flow10	57.635	115.565	95.932	121.412	133.42	99.995

Since this technique takes allocated time from the non-critical traffic, it is important to observe the behavior of such traffic in this scenario. Using the same topology from Figure 28, we were able to execute the experiment.

Figures 36 and 37 present the latency graph of the non-critical traffic with 5% and 10% decrease respectively. The impacts of the window reduction become apparent and are reinforced with the results comparison presented in Table 9.

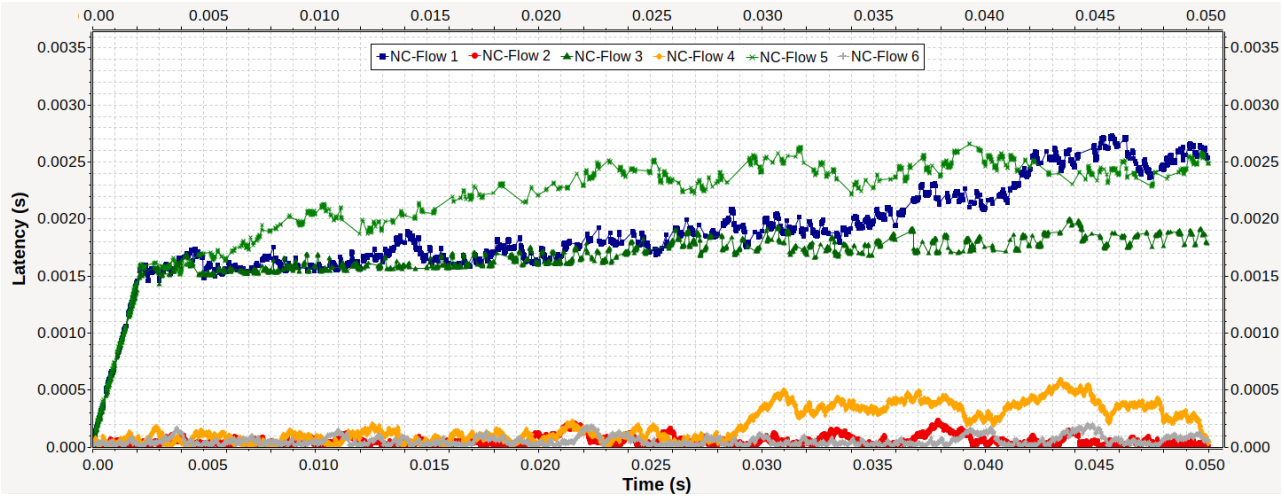

Figure 36: Grossly modified schedule - 5% decrease of non-critical window.

Lastly, we also analyzed the latency behavior of the non-critical traffic using the window decrease in scenarios with Switch Overload. Figures 38 and 39 show the latency

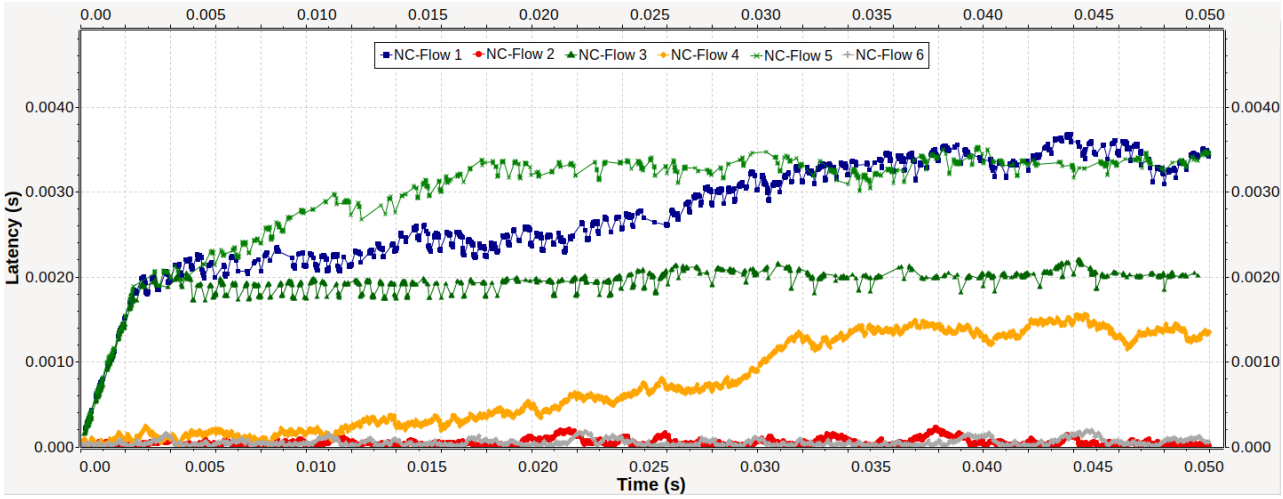

Figure 37: Grossly modified schedule - 10% decrease of non-critical window.

Table 9: Comparison of non-critical traffic results.

	Scenario					
	Original Schedule		5% Window Decrease		10% Window Decrease	
	Latency (μ s)	Jitter (μs)	Latency (μ s)	Jitter (μs)	Latency (μ s)	Jitter (μs)
NC-Flow1	1577.441	255	1907.569	399.348	2763.56	642.403
NC-Flow2	56.19	41.5	56.19	41.45	56.18	41.45
NC-Flow3	1559.193	242.209	1666.465	233.257	1920.035	264.351
NC-Flow4	71.984	60.349	199.274	151.215	751.704	521.303
NC-Flow5	1535.527	265.304	2169.463	457.536	2998.912	627.478
NC-Flow6	56.849	39.412	56.852	39.534	56.852	39.534

graph, and unexpectedly, the number of latency peaks is considerably lower than the ones seen in Figure 30. However, the general latency of these flows still worse, as shown in Table 10.

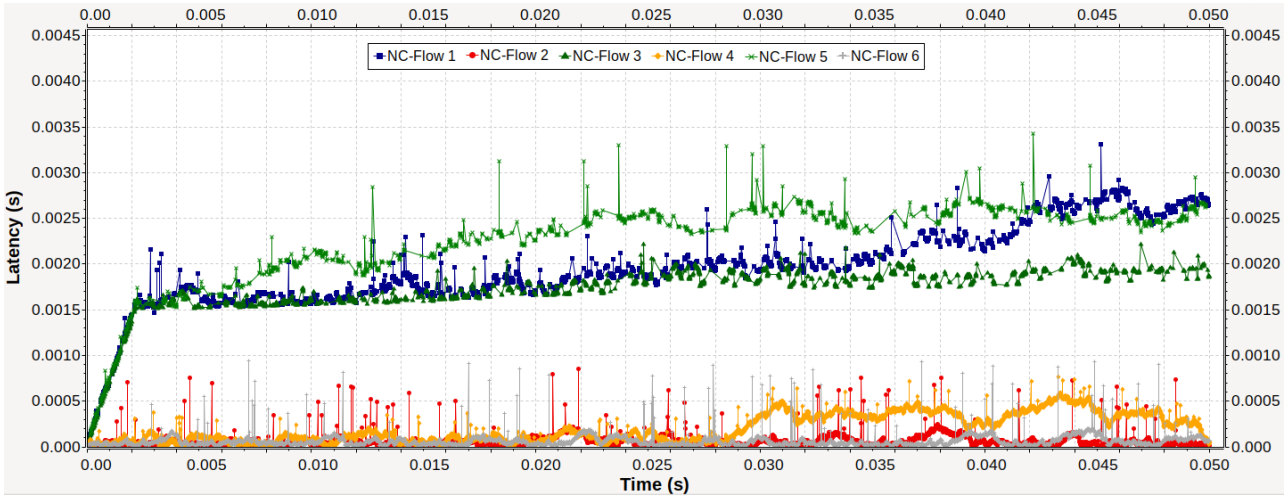

Figure 38: 5% decrease of non-critical window with switch overload.

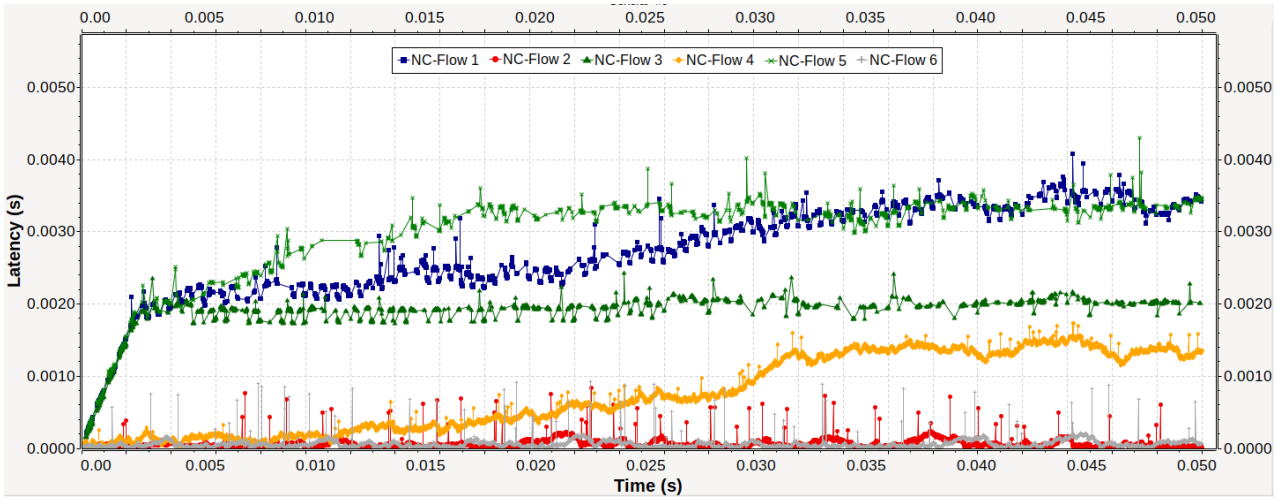

Figure 39: 10% decrease of non-critical window with switch overload.

Table 10: Comparison of non-critical traffic with switch overload results.

	Scenario					
	Original Schedule With Switch Overload and Protection Band		5% Window Decrease With Switch Overload		10% Window Decrease With Switch Overload	
	Latency (μ s)	Jitter (μs)	Latency (μ s)	Jitter (μs)	Latency (μ s)	Jitter (μs)
NC-Flow1	1777.86	368.057	1958.93	418.808	2697.294	621.43
NC-Flow2	61.654	57.094	62.168	59.288	62.302	57.816
NC-Flow3	1706.384	297.685	1739.87	259.291	1911.813	260.938
NC-Flow4	1275.572	96.723	200.682	151.271	756.666	52.243
NC-Flow5	1791.672	328.891	2261.856	457.536	3027.85	628.92
NC-Flow6	63.342	67.061	63.488	64.67	63.133	63.724

5.6 Schedule Checker

With the checker mechanism we implemented in TSNsched, we are now capable to validate the integrity of the generated schedule in design-time, without the use of simulation tools. To clarify, this is a preliminary validation and does not not make the evaluations done with simulation tools obsolete.

We can validate the schedule and log files generate by TSNsched in seconds using the checker, otherwise this task would take hours to be done manually. Using the typologies and scenarios presented in this work as well as additional test scenarios, we ran the checker using the criteria presented in Section 4.4.3.

In all our executions no error was triggered, proving the efficiency of TSNsched. However, warnings were generated. These warning be triggered in different cases, i.g. multiple node send traffic at the same time, the priority windows are too tight, etc. These cases are not considered errors, since the network can still work, but the strictness of the generated schedule can be harmful, thus the warning is generated.

Previously in this work we mentioned a limitation of the redundancy standard IEEE 802.1CB. If the network topology is not well connected, offering multiple paths for the Frame Replication and Elimination mechanism, this TSN standard becomes obsolete. With this in mind, we added a new validation on TSNsched.

The new validation checks the connectivity of the switches. Ideally, the network should be fully connected to take full advantage of IEEE 802.1CB standard. To be fully connected, the network must have $\sum_{k=1}^{N-1} k$ connections (where N is the number of switches). If the network is not well connected, an error is triggered, and then the flows are analyzed, if a flow have a switch with few connections (less than N-1, where N is the number of switches) in their path, that flow is flagged as at risk. Figure 40 depicts the result of the connectivity validation on the topology used in this work, and as you can see, the network is not well connected, putting most of the flows at risk of faults such as Link failure and Transmitter/Receiver failure.

```
Switch connections:
    switch0.ethg[0] <--> C <--> switch2.ethg[0];


    switch1.ethg[0] <--> C <--> switch2.ethg[1];


    switch2.ethg[2] <--> C <--> switch3.ethg[0];

RISK DETECTED!
This topology have only 3 connections. The ideal number of connections for the IEEE802.1 CB standard
work and take full advantage of the FRER (Frame replication and Elimination for Reliability) mechanism
should be 6. The following data flows may be compromised in the case of a network fault:

    The flow1goes through the switch0. Thus, a risk was detected.
    The flow2goes through the switch0. Thus, a risk was detected.
    The flow3goes through the switch0. Thus, a risk was detected.
    The flow4goes through the switch1. Thus, a risk was detected.
    The flow5goes through the switch1. Thus, a risk was detected.
    The flow6goes through the switch1. Thus, a risk was detected.
    The flow7goes through the switch1. Thus, a risk was detected.
    The flow8goes through the switch1. Thus, a risk was detected.
    The flow13goes through the switch0. Thus, a risk was detected.
    The flow14goes through the switch3. Thus, a risk was detected.
    The flow15goes through the switch3. Thus, a risk was detected.
    The flow16goes through the switch3. Thus, a risk was detected.
    The flow17goes through the switch3. Thus, a risk was detected.
    The flow18goes through the switch3. Thus, a risk was detected.
```

Figure 40: Switch connectivity validation.

6 Final Considerations and Future Works

In this study, our primary focus was on advancing the capabilities of the NeSTiNg simulation model, particularly in facilitating fault-injection procedures. By augmenting the NeSTiNg simulation model, we paved the way for a comprehensive analysis of how these injected faults impact TSN schedules and their overall reliability. This endeavor was greatly aided by the integration of the enhanced simulation model with the automated schedule generator TSNsched, allowing us to conduct in-depth examinations of fault scenarios and their implications.

Furthermore, recognizing the critical need for fault tolerance in real-world networking environments, we devised and implemented a novel fault-tolerant technique within the Nest-Sched tool. This innovation empowers seamless recovery from Switch Overload scenarios, significantly enhancing the robustness and resilience of TSN networks without necessitating disruptive re-scheduling processes.

In addition to fortifying the Nest-Sched tool, we enriched the schedule generator TSNsched with an array of new validation mechanisms. Leveraging a meticulously crafted set of criteria, we systematically confirmed the feasibility of generated schedules, thereby ensuring their suitability for deployment in practical settings. Moreover, we introduced advanced fault-tolerant mechanisms during the design phase, enabling proactive evaluation of network connectivity and facilitating comprehensive risk assessments for data flows.

The culmination of our efforts materialized in the form of two impactful papers presented at the esteemed Brazilian Symposium on Telecommunications and Signal Processing (SBrT: https://www.sbrt.org.br). These publications, cited as [48] and [49], not only underscore the significance of our contributions but also serve as a testament to our ongoing commitment to advancing the field. As we continue to push the boundaries of innovation, our endeavors remain steadfastly focused on developing novel fault-tolerance techniques and further refining the tools essential for the generation and validation of TSN schedules.

References

[1] C. Simon, M. Maliosz and M. Mate, "Design Aspects of Low-Latency Services with Time-Sensitive Networking," in IEEE Communications Standards Magazine, vol. 2, no. 2, pp. 48-54, June 2018.

[2] W. Steiner, S. S. Craciunas and R. S. Oliver, "Traffic Planning for Time-Sensitive Communication," in IEEE Communications Standards Magazine, vol. 2, no. 2, pp. 42-47, June 2018.

[3] IEEE, *Time Sensitive Networking Task Group*, http://www.ieee802.org/1/pages/tsn.html

[4] B. Simon, U. Ecehan. "Time-Sensitive Networking: From Theory to Implementation in Industrial Automation". TTTech and Intel. 2015.

[5] S. S. Craciunas, R. S. Oliver, M. Chmelík, and W. Steiner. 2016. Scheduling Real-Time Communication in IEEE 802.1Qbv Time Sensitive Networks. In Proceedings of the 24th International Conference on Real-Time Networks and Systems (RTNS '16). Association for Computing Machinery, New York, NY, USA, 183–192. https://doi-org.ez15.periodicos.capes.gov.br/10.1145/2997465.2997470

[6] A. C. T. d. Santos, B. Schneider and V. Nigam, "TSNSCHED: Automated Schedule Generation for Time Sensitive Networking". 2019. Formal Methods in Computer Aided Design (FMCAD), San Jose, CA, USA, 2019, pp. 69-77.

[7] M. Pahlevan and R. Obermaisser, "Redundancy Management for Safety-Critical Applications with Time Sensitive Networking," 2018 28th International Telecommunication Networks and Applications Conference (ITNAC), Sydney, NSW, Australia, pp. 1-7, 2018.

[8] J. Falk et al., "NeSTiNg: Simulating IEEE Time-sensitive Networking (TSN) in OMNeT++," 2019 International Conference on Networked Systems (NetSys), Munich, Germany, 2019, pp. 1-8.

[9] Audio Video Bridging Task Group. IEEE 802.1. <https://ieee802.org/1/pages/avbridges.html>.

[10] IEEE, *IEEE 802.1 Working Group*, https://1.ieee802.org

[11] IEEE, *Time Sensitive Networking Task Group*, http://www.ieee802.org/1/pages/tsn.html

[12] Deterministic Ethernet, https://www.tttech.com/explore/deterministic-ethernet

[13] R. Hummen, S. Kehrer, L. Wüsteney, TSN - Time Sensitive Networking White Paper, 2019 Hirschmann a Belden brand.

[14] M. Ashjaei, L. Murselović, S. Mubeen, "Implications of Various Preemption Configurations in TSN Networks," in IEEE Embedded Systems Letters, vol. 14, no. 1, pp. 39-42, March 2022, doi: 10.1109/LES.2021.3103061.

[15] Reusch, Niklas, Zhao, Luxi, Craciunas, Silviu S., Pop, Paul. "Window-Based Schedule Synthesis for Industrial IEEE 802.1Qbv TSN Networks" 2020 16th IEEE International Conference on Factory Communication Systems (WFCS). pp. 1-4.

[16] IEEE Standard for Local and Metropolitan Area Network –Bridges and Bridged Networks, IEEE Std. 802.1Q-2018, pp. 1-1993, Rev. IEEE Std. 802.1Q-2014, 2018.

[17] IEEE Standard for Local and Metropolitan Area Networks - Virtual Bridged Local Area Networks Amendment 12: Forwarding and Queuing Enhancements for Time-Sensitive Streams, IEEE Std. 802.1Qav-2009, pp. C1-72, 2010.

[18] S. Kehrer, O. Kleineberg and D. Heffernan, "A comparison of fault-tolerance concepts for IEEE 802.1 Time Sensitive Networks (TSN)," Proceedings of the 2014 IEEE Emerging Technology and Factory Automation (ETFA), Barcelona, Spain, 2014, pp. 1-8, doi: 10.1109/ETFA.2014.7005200.

[19] IEEE Standard for Local and Metropolitan Area Networks–Bridges and Bridged Networks–Amendment 28: Per-Stream Filtering and Policing, Standard 802.1Qci-2017, 2017, pp. 1–65

[20] Z3 Solver, https://github.com/Z3Prover/z3

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

[22] Objective Modular Network Testbed in C++, *OMNeT++*, [Online]. Disponível: https://omnetpp.org/

[23] INET Framework, [Online]. Disponível: https://inet.omnetpp.org/

[24] Y. Seol, D. Hyeon, J. Min, M. Kim and J. Paek, "Timely Survey of Time-Sensitive Networking: Past and Future Directions," in IEEE Access, vol. 9, pp. 142506-142527, 2021, doi: 10.1109/ACCESS.2021.3120769.

[25] E. Baccarelli, P. G. V. Naranjo, M. Scarpiniti, M. Shojafar and J. H. Abawajy, "Fog of Everything: Energy-Efficient Networked Computing Architectures, Research Challenges, and a Case Study," in IEEE Access, vol. 5, pp. 9882-9910, 2017, doi: 10.1109/ACCESS.2017.2702013.

[26] L. Monostori, B. Kádár, T. Bauernhansl, S. Kondoh, S. Kumara, G. Reinhart, O. Sauer, G. Schuh, W. Sihn, K. Ueda, Cyber-physical systems in manufacturing, CIRP Annals, Volume 65, Issue 2, 2016, Pages 621-641, ISSN 0007-8506, https://doi.org/10.1016/j.cirp.2016.06.005.

[27] N. G. Nayak, F. Dürr and K. Rothermel, "Incremental Flow Scheduling and Routing in Time-Sensitive Software-Defined Networks," in IEEE Transactions on Industrial Informatics, vol. 14, no. 5, pp. 2066-2075, May 2018, doi: 10.1109/TII.2017.2782235.

[28] J. Falk, F. Dürr and K. Rothermel, "Exploring Practical Limitations of Joint Routing and Scheduling for TSN with ILP," 2018 IEEE 24th International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA), Hakodate, Japan, 2018, pp. 136-146, doi: 10.1109/RTCSA.2018.00025.

[29] Q. Yu and M. Gu, "Adaptive Group Routing and Scheduling in Multicast Time-Sensitive Networks," in IEEE Access, vol. 8, pp. 37855-37865, 2020, doi: 10.1109/ACCESS.2020.2974580.

[30] D. Hellmanns, A. Glavackij, J. Falk, R. Hummen, S. Kehrer and F. Dürr, "Scaling TSN Scheduling for Factory Automation Networks," 2020 16th IEEE International Conference on Factory Communication Systems (WFCS), Porto, Portugal, 2020, pp. 1-8, doi: 10.1109/WFCS47810.2020.9114415.

[31] M. H. Farzaneh, S. Kugele and A. Knoll, "A graphical modeling tool supporting automated schedule synthesis for time-sensitive networking," 2017 22nd IEEE International Conference on Emerging Technologies and Factory Automation (ETFA), Limassol, Cyprus, 2017, pp. 1-8, doi: 10.1109/ETFA.2017.8247599.

[32] W. Steiner, "An Evaluation of SMT-Based Schedule Synthesis for Time-Triggered Multi-hop Networks," 2010 31st IEEE Real-Time Systems Symposium, San Diego, CA, USA, 2010, pp. 375-384, doi: 10.1109/RTSS.2010.25.

[33] Communication over Real-Time Ethernet Group, *CoRE4INET Framework*, https://core-researchgroup.de/projects/simulation.html

[34] A. Bergstrom. Automatic generation of network configuration in simulated time sensitive networking (TSN) application. Master's thesis, Malardalen University, Vasteras, Sweden, 2020

[35] B. Houtan, A. Bergström, M. Ashjaei, M. Daneshtalab, M. Sjödin and S. Mubeen, "An Automated Configuration Framework for TSN Networks," 2021 22nd IEEE International Conference on Industrial Technology (ICIT), Valencia, Spain, 2021, pp. 771-778, doi: 10.1109/ICIT46573.2021.9453628.

[36] I. Koren, C.M. Krishna, Fault-Tolerant Systems, Morgan Kaufmann, 2020

[37] M. Kim, J. Min, D. Hyeon and J. Paek, "TAS Scheduling for Real-Time Forwarding of Emergency Event Traffic in TSN," 2020 International Conference on Information and Communication Technology Convergence (ICTC), Jeju, Korea (South), pp. 1111-1113, 2020.

[38] M. Kim, D. Hyeon and J. Paek, "eTAS: Enhanced Time-Aware Shaper for Supporting Nonisochronous Emergency Traffic in Time-Sensitive Networks," in IEEE Internet of Things Journal, vol. 9, no. 13, pp. 10480-10491, 1 July1, 2022.

[39] M. Ashjaei, M. Sjödin, S. Mubeen, A novel frame preemption model in TSN networks, Journal of Systems Architecture, Volume 116, 2021, 102037, ISSN 1383-7621, https://doi.org/10.1016/j.sysarc.2021.102037.

[40] W. Kong, M. Nabi and K. Goossens, "Run-Time Recovery and Failure Analysis of Time-Triggered Traffic in Time Sensitive Networks," in IEEE Access, vol. 9, pp. 91710-91722, 2021, doi: 10.1109/ACCESS.2021.3092572.

[41] Z. Feng, Z. Gu, H. Yu, Q. Deng and L. Niu, "Online Rerouting and Rescheduling of Time-Triggered Flows for Fault Tolerance in Time-Sensitive Networking," in IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, vol. 41, no. 11, pp. 4253-4264, Nov. 2022, doi: 10.1109/TCAD.2022.3197523.

[42] I. Álvarez, J. Proenza, M. Barranco and M. Knezic, "Towards a time redundancy mechanism for critical frames in time-sensitive networking," 2017 22nd IEEE International Conference on Emerging Technologies and Factory Automation (ETFA), 2017, pp. 1-4, doi: 10.1109/ETFA.2017.8247721.

[43] R. M. Silva, A. C. T. Santos, V. Nigam, F. Iguatemi E. Fonseca, "Nest-Sched: Validating TSN Traffic Scheduling Through Simulation", XI Conferência Nacional em Comunicações, Redes e Segurança da Informação, pp. 70-72, 2021.

[44] Chockler, Hana, Weissenbacher, Georg, "Computer aided verification", Springer International Publishing, Lecture notes in computer science, 2018.

[45] A. Machiry, C. Spensky, J. Corina, N. Stephens, C. Kruegel and G. Vigna, "DR. CHECKER: A Soundy Analysis for Linux Kernel Drivers", 26th USENIX Security Symposium (USENIX Security 17), pp. 1007-1024, 2017.

[46] I. Miren, E. Leire, L. Felix and S. Goiuria, "CRESCO Framework and Checker: Automatic generation of Reflective UML State Machine's C++ Code and Checker", 2020 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW), pp. 25-30, 2020.

[47] T. Kapus, “Using PRISM model checker as a validation tool for an analytical model of IEEE 802.15.4 networks”, Simulation Modelling Practice and Theory, pp. 367-378, 2017.

[48] K. L. Morais, R. M. Silva, A. C. Santos, V. Nigam, I. E. Fonseca, “TSN: Checker tool for output validation”, XL Simpósio Brasileiro de Telecomunicações e Processamento de Sinais (SBrT2022), doi: 10.14209/sbrt.2022.1570816056, 2022

[49] R. M. Silva, A. C. Santos, I. E. Fonseca, V. Nigam, “A Study on TSN Scheduling Robustness”, XLI Simpósio Brasileiro de Telecomunicações e Processamento de Sinais (SBrT2023), doi: 10.14209/sbrt.2023.1570923319, 2023