A Study on TSN Scheduling Robustness

Renan M. Silva, Aellison C. T. Santos, Iguatemi E. Fonseca e Vivek Vigam

***Abstract*— Time Sensitive Networking (TSN) provides high performance deterministic communication using time scheduling. In theory, the rigor of a TSN schedule is the key to achieving deterministic communication; however, real devices are prone to errors. TSN-based applications require both fault-tolerance and end-to-end latency guarantee. This paper identifies the limitations of the TSN scheduling method, enhancing the simulation model NeSTiNg to enable fault-injection and integrating it to the schedule generator tool-chain TSNsched.**

***Keywords*— Time Sensitive Networking, Simulation, Fault-Tolerance**

I. Introduction

Modern industrial systems have several traffic types, such as, time-critical, best-effort, and audio/video. Each of these traffic flows have different priorities and requirements making the network design a complicated task [1]. More and more fields of the industry require deterministic 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 [3]. The 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), and traffic scheduling through time (IEEE 802.1Qbv) [4]. The Time Aware Shaper (TAS, IEEE 802.1Qbv) is the scheduling technique proposed by TSN. To create a TSN 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 the schedule is pre-calculated, static values are used to represent relevant parameters, such as packet size, periodicity, number of devices, route, transmission and processing delay

When we work with real scenarios, however, devices may not work in a constant stable manner as they can present errors and inconsistencies. For example, if a TSN end-device or switch fails, all the scheduling and latency guarantee is at risk. Thus, fault-tolerance aspects must be taken into consideration when creating a TSN schedule.

In order to identify and evaluate the robustness limits of TSN scheduling, this paper uses TSNsched [5] to generate the network schedule according to TSN constraints, and the simulation model NeSTiNg [6] to simulate an environment where the devices may present issues and analyze the behavior of the network.

Renan M. Silva, Aellison C. T. Santos, Iguatemi E. Fonseca e Vivek Nigam, Programa de Pós-graduação em Informática, UFPB, João Pessoa - PB, e-mails: piumoreiras@gmail.com; cassimiroaellison@gmail, iguatemi@ci.ufpb.br, vivek.nigam@gmail.com. Este trabalho foi parcialmente financiado pelo CNPq e pela CAPES.

The main contributions of this paper are:

- **Simulation Model Enhancement:** We customized the simulation model NeSTiNg, enabling fault-injection. This enhanced simulation model allows the user to emulate different faults and easily configure their associated parameters and probabilities.
- **Tool-Chain Integration:** We implemented the faults on NeSTiNg and integrated with the TSNsched tool-chain. The faults can be added automatically using TSNsched output and be loaded directly to NeSTiNg.
- **TSNsched Validation:** We validate TSNsched output in different scenarios, analyzing the effects of failures and the tool limitations.

The outline of the paper is as follows: Section II presents a brief overview of the Time Aware Shaper, tools used in the study, as well as the related work. Section III goes in depth on the possible network faults and how we enhanced the simulation model to emulate these faults. Section IV evaluates and compare the network performance with and without faults. Section V concludes the paper and points to direction of future work.

II. Background

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) [7], this mechanism is responsible for scheduling traffic into eight different priorities. Fig. 1 illustrates how this mechanism works. Each port on a TSN switch have eight priority queues and eight forwarding gates. When a packet arrives at a switch, it is processed and goes through a filter, often called the *"Switching Fabric"*. This Priority Filter will determine the packet priority (attribute found in the packet header) and then direct it to its particular Priority Queue.

Each priority queue follows a FIFO (*First In First Out*) policy, but packets from that queue can only be sent when their respective forwarding gate is open. What determines the status of the forwarding gate is the scheduling table used by that gate *(GCL - Gate Control List)*. S This scheduling mechanism aims to protect and ensure the transmission periods of different priorities. But due to the strict constraints of TAS and the fact the schedule set on the GCL is pre-calculated, the end-to-end latency cannot be guaranteed if a device were to present faults or a new traffic flow were introduced [8].

A. Tools

TSNSCHED [5] is an application that uses the Z3 SMT (*Satisfiability Modulo Theories*) Solver [9] to generate the

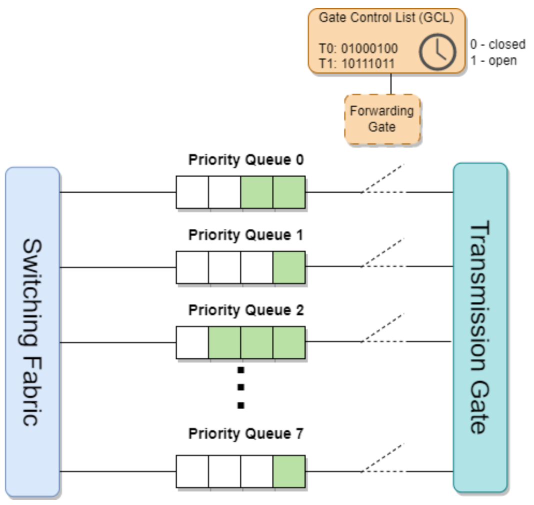

Fig. 1. Time-Aware Shaper. [2]

network schedule given a configuration file with the network topology, latency and jitter requirements as input. This tool is used in this paper to generate the schedule needed for our experiment.

OMNeT++ (*Objective Modular Network Testbed in C++*) [11] is an object-oriented event-based simulator, usually used to build network simulation models, having an IDE based in the Eclipse platform.

The INET-Framework [10] is an open source set of modules and protocols for OMNeT++ capable of simulating wired, wireless and mobile networks.

NeSTiNg [6] is a simulation model for OMNeT++, using the INET-Framework and enhancing it adding components to enable TSN simulation. In the next section we go through or contributions to the simulation model, explaining the components added to enable simulation of TSN networks in unstable scenarios. In a previous work, we performed a preliminary analysis of simulations scenarios using theses tools [16].

B. *Related Work*

Emergency traffic have the highest priority and may occur sporadically without prior notice. The support for asynchronous emergency traffic in TSN was first introduced by [13]. In this work, the authors offer a solution in the form of a *Protection Band* working as a fail safe mechanism. The paper [14] is an extension of this work, where 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, so if multiple faults were to happen at the same time, the solution proposed could not be applied.

Similar to our work, [15] uses simulation tools to check the limitations of TSN. More precisely, the authors show the limitations of the TSN packet preemption standard and propose a new preemption model to improve the maximum response time of high priority frames. A shortcoming of this paper is the narrow view for improvements, the frame preemption model have a limit to its usefulness and other faults should be taken into consideration.

Most of the studies found in the state of the art consider TSN faults from the scheduling perspective. Narrowing their point of view to the introduction of new traffic flows and send window optimization. Our work approach TSN shortcomings considering failures in network devices, allowing a broader set of faults and experiments. Moreover, our work is integrated to the TSNsched tool-chain, enabling schedule validation and fault injection in a semi-automatic way.

III. 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. Being capable to attend the requirements of different types of traffic, increase the demand for reliability.

TSN have sub-standards dedicated to achieve this reliability. 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, having in mind the elimination of duplicated frames at the destination [12].

Even though TSN has a certain level of fault-tolerance, these techniques can only work if a network is in prime conditions. If the network present a certain level of instability (e.g. network overload), the latency guarantee will be lost.

A. *Network Faults*

After taking into consideration the scheduling mechanisms and fault-tolerance build into TSN, we concluded the following faults represent the most common real-life scenarios of inconsistency for a TSN network [12].

- **Switch Overload:** This fault happens when a switch takes too long to process the packet, making the packet arrive late at the priority queue and missing its send window.
- **Link Failure:** This error happens when a packet is dropped at the switch port. This occurs when a switch have a link failure making the transmission unreliable.
- **Transmitter Failure:** 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 right receiver or disturb the traffic flow of other priority.
- **Receiver Failure:** This fault happens when a receiving device fails to sort the received packets in the right order, dropping useful packets by mistake.

B. *Simulation*

To analyze the scenarios where TSN robustness is pushed to its limits, we made modifications to the simulation model NeSTiNg, creating new modules to emulate faults. [1]

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

To emulate the Switch Overload fault, we created a switch with a new delayer. On NeSTiNg, the switch delayer works as a mean to emulate the time it takes to process the packet. So we created a new delayer and turned the static variable used to define the processing delay into a seed to a probability distribution. The probability distribution chosen for this work was the Uniform Distribution, 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, a function with a probability distribution is ran using the seed provided to determine if the packet is received or not.

The end devices in NeSTiNg use the same module to generate and receive traffic. To emulate the Transmitter Failure the traffic module uses a probability distribution to check if it will send the packet with the right information. When receiving a packet, he 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.

IV. Experiments and Evaluation

In this section, we describe the simulation setup, going through the network specification and expected latency. Then we introduce the faults on the network and evaluate the impact they had on the overall latency.

A. Simulation Setup

We use the tool TSNsched to generate the schedule for the network depicted in Fig. 2. The chosen topology have 10 end-devices, 4 switches and 10 traffic flows with different priorities, packet length and periodicity. The details about the network can be found on Table I. 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 I
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

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 [16] to generate the simulation files required by NeSTiNg automatically. A visual representation of the tool-chain can be found in Fig. 3.

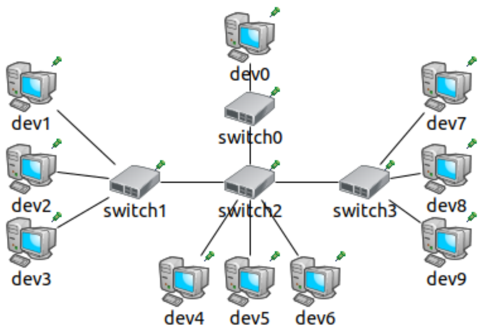

Fig. 2. Simulated Network.

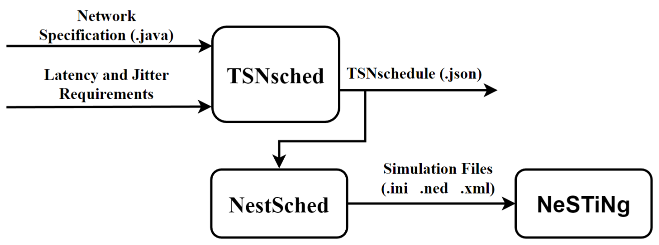
Network Specification (.java)
Latency and Jitter Requirements
TSNsched
TSNschedule (.json)
NestSched
Simulation Files (.ini .ned .xml)
NeSTiNg

Fig. 3. Tool-Chain.

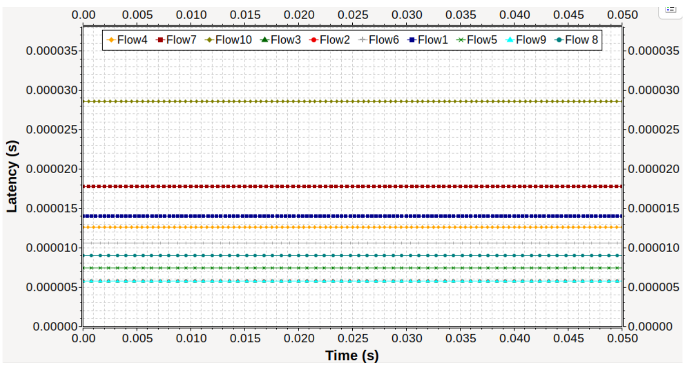

Fig. 4. Network Expected Latency.

B. 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 Fig. 4 represents the network latency if it were working in perfect conditions. As you can see, all the flows remain stable, meeting the specified constraints.

Now that we have the schedule and the necessary configuration files for the simulation, we can inject the faults. As mentioned in Section 3, new components were created to enhance NeSTiNg simulation model, each new component uses a uniform probability distribution to determine if it will behave normally or present issues. 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 Fig. 5.

```
**.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      # %
```

Fig. 5. Fault Configuration.

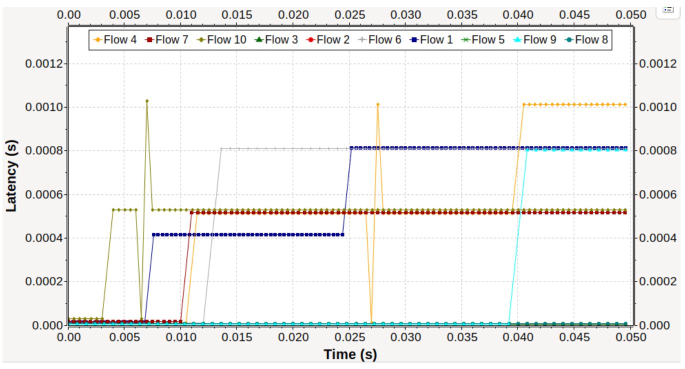

Fig. 6. Switch Overload.

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 seed 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. The resulting latency can be seen in Fig. 6.

The harmful strictness of the TSN scheduling mechanism is shown on this scenario. Applying a fault probability of 1% to the switches was enough to disrupt the entire schedule, increasing the overall latency and jitter.

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, let's say the network stability has worsened and all the switches are operating under critical conditions. Instead of a cascading delay, the affected flows will present a worse variation, as depicted in Fig. 7.

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 Fig. 8, after some time, the fault is triggered and the flows directed to *dev0* are interrupted. This network topology (Fig. 2) was used to show

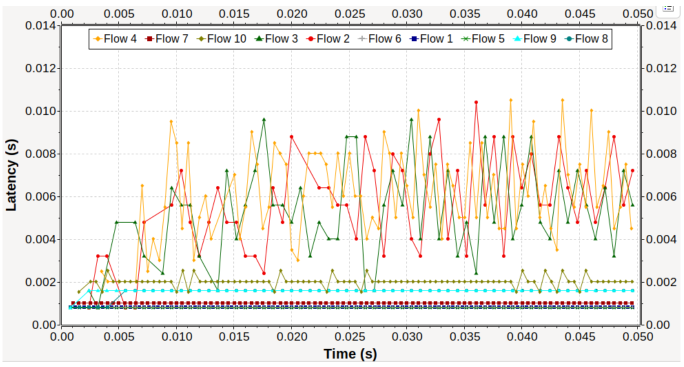

Fig. 7. Switch Overload. Critical Case

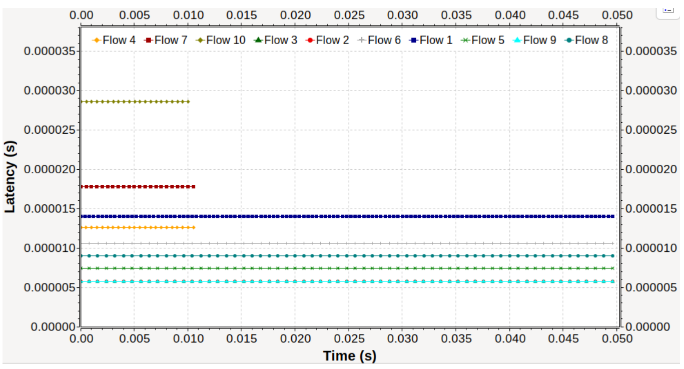

Fig. 8. Link Failure

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.

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.

Fig. 9 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.

C. Latency

The latency results of all scenarios are summarized in Table II. 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; 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

TABLE II
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

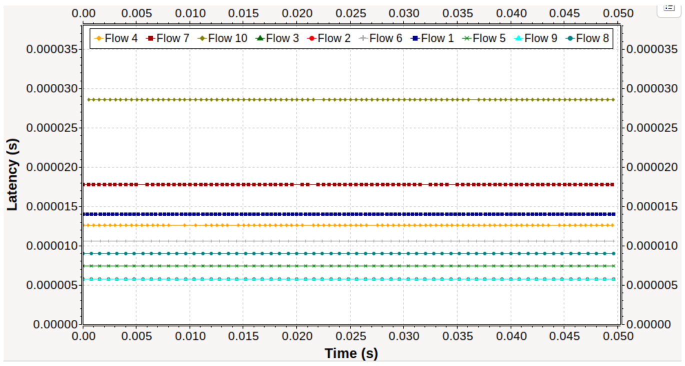

Fig. 9. Transmitter/Receiver Failure

traffic flows are still being affected since the packets are not arriving at their destinations and information is lost.

V. Conclusion and Future Work

In this paper, we extended the simulation model NeSTiNg to support fault-injection. Using this enhanced simulation model and the automated schedule generator TSNsched, we were able to analyze how these faults affect the TSN schedule and reliability. There are different types of faults and each of them can affect the network in a different manner. When we speak about TSN, the most noticeable fault is related to schedule disruption, but network robustness goes beyond that. Traffic planning and network modeling are also critical parts, as shown in this paper.

TSN have the minimum fault-tolerance mechanisms, relying on network planning, controlled and stable scenarios. The results found in this paper show that even with those mechanisms is still possible to disrupt the well crafted schedule. Most of the fault-tolerant studies found on the state-of-the-art deal with run-time solutions. The re-creation of the schedule and configuration on devices may be costly. Thus, using the results found on this paper, our next step is to come up with new fault-tolerant mechanisms for TSN on design-time, enhancing the schedule generator TSNsched and integrating fault-tolerant aspects to the open-source tool chain.

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, last accessed on August 5th 2023

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

[5] 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.

[6] 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.

[7] Reusch, Niklas and Zhao, Luxi and Craciunas, Silviu S. and 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.

[8] 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.

[9] Z3 Solver, https://github.com/Z3Prover/z3 , last accessed on August 5th 2023

[10] INET Framework, https://inet.omnetpp.org , last accessed on August 5th 2023

[11] Objective Modular Network Testbed in C++, OMNeT++, https://omnetpp.org/ , last accessed on August 5th 2023

[12] 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.

[13] 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.

[14] 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.

[15] 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,

[16] Renan M. Silva, Aellison C. T. Santos, Vivek 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.