IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received 19 July 2022, accepted 25 September 2022, date of publication 4 October 2022, date of current version 19 October 2022.

Digital Object Identifier 10.1109/ACCESS.2022.3211969

METHODS

Methodology and Infrastructure for TSN-Based Reproducible Network Experiments

MARCIN BOSK[1], FILIP REZABEK[1], KILIAN HOLZINGER[1], ANGELA GONZALEZ MARINO[2], ABDOUL AZIZ KANE[2], FRANCESC FONS[2], (Senior Member, IEEE), JÖRG OTT[1], AND GEORG CARLE[1]

[1]Department of Informatics, Technical University of Munich, 85748 Garching bei München, Germany
[2]Huawei Technologies Düsseldorf GmbH, 40549 Düsseldorf, Germany

Corresponding author: Marcin Bosk (bosk@in.tum.de)

Marcin Bosk and Filip Rezabek contributed equally to this work.

ABSTRACT Time-Sensitive Networking (TSN) is a set of standards offering bounded latency and jitter, low packet loss, and reliability for Ethernet-based systems while allowing best-effort and real-time traffic to coexist. Domains that use TSN include intra-vehicular networks (IVNs), aerospace, professional audio-video solutions, and smart manufacturing. All these areas shift towards Ethernet due to its scalability, throughput, easy to develop applications, and affordability to produce in a large scale. In this work, we devise a methodology that introduces a workflow comprising several steps to assess TSN in various domains. The first step defines requirements and assesses which real-time traffic is present within a given domain. The second step focuses on configuration of a representative TSN-based network. The step proceeds with performance evaluation of different TSN standards in the chosen configuration(s). The third – optional – step supports optimizing the system to fulfill the identified requirements. The methodology is generalized by assessing the various TSN domains and finding their commonalities. As a result, we see the methodology can be applied to other TSN solutions. We provide a detailed case study for the domain of IVNs, from which the methodology is derived. We summarize the key requirements, systematically analyze IVNs traffic patterns for real-time and best effort traffic, and evaluate the performance of crucial TSN standards recommended by the IEEE 802.1DG Automotive Profile. The methodology builds on top of infrastructure framework, EnGINE, that offers an environment for reproducible and scalable TSN experiments and relies on commercial off the shelf hardware and open-source solutions. The framework allows to evaluate various standards and identify suitable topologies with focus on Layer 2 solutions. Using EnGINE, we evaluated the various traffic patterns and their corresponding TSN configurations and identified if and how the IVN requirements can be fulfilled.

INDEX TERMS Deterministic networking, experiment infrastructure, in-vehicular networking, networking experiments, time-sensitive networking, time-sensitive networking methodology, reproducible experiments.

I. INTRODUCTION

Today's networks are pursuing an ever greater data throughput, accompanied by an increase in their scale and complexity [1]. Operators of these systems are keen on lowering the costs and shift towards Ethernet based solutions even in more specialized domains [2]. However, with all its benefits, this networking standard does not, by default, offer deterministic behavior and guarantees on bounded latency and jitter, low packet loss, and reliability. These properties are especially relevant for the real-time and time-sensitive systems, such as Intra-Vehicular Networks (IVNs), aerospace applications, smart manufacturing, professional audio and video, and many others.

The Time Sensitive Networking (TSN) capabilities and performance have been an active research subject, with many evaluations being conducted in simulation environments. Using such environments brings multiple advantages. The designed experiments are easy to reproduce and configure, offer high flexibility, and fast development cycle. The main

The associate editor coordinating the review of this manuscript and approving it for publication was Yougan Chen.

disadvantage is simulations' deviation from realistic system behavior, as real deployment artifacts such as clock deviation or operating system overhead are lacking. On the other hand, for real deployments, we need to consider what hardware and software should be used to fulfill given real-time requirements, evaluate the impact of the various artifacts present in the system, and get an understanding of such shortcomings.

Therefore, with this work we introduce a methodology that aims to provide a structural approach to assess system behavior, recommend how to utilize and configure TSN standards, and identify ways of performance optimization to achieve the real-time guarantees. Our approach combines simulations' advantages while running in a physical topology containing machines emulating network switches and constrained devices within various domains where TSN is required. In the following, we aim to provide guidelines on how the new generations of networks based on Ethernet can be assessed and validated, considering their respective requirements.

The aforementioned methodology consists of two elements: an evaluation framework, and network performance assessment guidelines with focus on system optimization to meet defined Key Performance Indicators (KPIs). The first component builds on top of already published results introducing the *EnGINE* framework (**En**vironment for **G**eneric **I**n-vehicular **N**etwork **E**xperiments) [3]. Even though the framework was originally designed for IVNs, with this work we show that it can be generalized and used for multiple other TSN domains. The framework brings high flexibility to network and data sources configuration. Moreover, *EnGINE* allows to monitor and record events for later evaluation or traffic re-play in the network to identify architectural limits. The experiments are easy to reproduce and configure. Furthermore, the framework enables emulation of possible malfunctions that can affect reliability, being a crucial aspect of real-time networked systems. *EnGINE* can introduce link failures and packet loss, to verify the system resilience.

With our methodology being based upon the *EnGINE* framework, our approach utilizes the Linux networking stack offering various queuing disciplines (qdiscs) configurations and TSN capable Commercial off-the-Shelf (COTS) hardware. We support the TSN standards recommended by various profiles, such as the IEEE P802.1DG TSN Profile for Automotive In-Vehicle Ethernet Communications [4], IEEE P802.1DP – TSN for Aerospace Onboard Ethernet Communications [5], and IEC/IEEE 60802 TSN Profile for Industrial Automation [6]. We place our focus on IEEE 802.1Qav [7], IEEE 802.1Qbv [8], and IEEE 802.1AS [9] standards with potential for extension and also inclusion of higher-layer mechanisms for time-sensitive network operation.

The second part of the methodology focuses on definition of individual actions needed to implement and deploy the aforementioned TSN standards within a given domain. The following steps are required:

- **S1** Definition of requirements and traffic patterns present in the given TSN domain
- **S2** Configuration and evaluation of TSN standards using the COTS hardware and open-source solutions
- **S3** System optimization to meet defined KPIs

Starting with **S1**, we need to understand what data and traffic patterns are present in such real-time networks, what priorities they have, and what KPIs they need to fulfill. Namely, we investigate packet loss, latency, and jitter with respect to packet sizes and spacing, and their priorities. To assess the data traffic patterns and their metrics, we use the recommendations presented by the AVNU Alliance for the individual Stream Reservation (SR) classes. AVNU Alliance target is to create an automotive ecosystem for the precise timing and low latency requirements for various applications using open standards. It stresses the importance of SR classes and their prioritization [10].

With the **S2**, we configure and evaluate the performance goals with respect to the defined requirements and assess the TSN standards in such scenarios.

Using the **S3**, we optimize the system to avoid unwanted artefacts and provide means for defining a suitable network configuration and interconnections, while also considering edge cases. These and other recommendations provide a good baseline for the network performance evaluation with fixed KPIs being the target for system optimization. Nevertheless, the scope of the work does not aim to provide a suitable solution, but rather required steps on how to achieve it.

We provide a detailed introduction and analysis of these actions within our methodology and their generalization in Section IV. This section also introduces a case study for the example domain of IVNs, where we go through the individual methodology steps and assess the network requirements. In addition, we introduce a sample topology that emulates a high-end IVN to which we apply the outcomes of general evaluations and identified traffic patterns. As a motivation for the applicability of TSN to IVN we consider use-cases such as shared mobility, vehicle-to-X communication, self-driving vehicles, and over-the-air upgrades. All of these use-cases require fast, secure, and reliable network. With this example, we want to show the applicability of our methodology to a smaller set of networks.

This work presents these Key Contributions (KCs):

- **KC1** Introduces methodology for assessment of TSN domains using an infrastructure for reproducible network experiments, e.g., EnGINE
- **KC2** Provides a case study using the defined methodology focusing on the domain of IVN, covering IVN requirements and traffic patterns definition
- **KC3** Provides detailed system optimization and TSN standards configuration and evaluation using the COTS hardware and open-source solutions

II. BACKGROUND

In this section, we introduce the requirements that IVNs in general and, by extension, the methodology presented in this work need to fulfill. Furthermore, we introduce all concepts

and technologies and build a robust methodology for IVN and TSN experiments. We also briefly introduce the *EnGINE* framework, the experimental environment we developed and used to evaluate the proposed methodology.

A. REQUIREMENTS FOR IN-VEHICULAR NETWORKS

The IVNs need to support a wide range of functions being executed within the scope of a vehicle. Since such a network carries information required for critical functionality, e.g., Advanced Driver Assistance Systems (ADAS) or brake-by-wire, it has to fulfill strict requirements for the network delay, jitter, packet loss, and robustness. Hence, in this work, we focus on the requirements of IVNs and combine those with the requirements of a hardware-based experimentation environment, our System Under Test (SUT). The SUT needs to enable IVN and TSN experiments as outlined in [3]. The experimental deployment itself needs to be capable of producing experiments which are, amongst others:

1) **Repeatable** – ease of experiment repetition within the same setup
2) **Reproducible** – ease of experiment result reproduction using the same setup
3) **Replicable** – ease of experiment result reproduction with a different setup offering same capabilities
4) **Configurable** – ease of configuration
5) **Autonomous** – no human intervention required
6) **Realistic** – use of real-world traffic patterns
7) **Scalable** – experiments with various complexities
8) **Diverse** – use of a wide variety of input formats

Furthermore, the SUT needs to fulfill the requirements of various IVN and TSN traffic types. We distribute these traffic types across various SR classes as defined in the recommendations of the AVNU alliance [10], based on the IEEE 8802-1BA [11] and the IEEE 1722-2016 [12] standards. An SR class summarizes the needs of traffic of a certain type. The resulting requirements on the performance of the network within the SUT are listed in Table 1. We determine that in order to enable IVN and TSN experiments, the hardware deployment needs to offer a latency less than 2ms. Furthermore, the end-to-end delay jitter, that is the difference in latency between two subsequent packets, needs to be kept under 125μs. This requirement needs to hold over 7 network hops for most critical traffic corresponding to SR class A. Less critical traffic does not require such low latencies and jitter. In the case of Best Effort (BE) traffic, no requirements are applicable. Each of the SR classes is mapped to a corresponding Priority Code Point (PCP) which can be used to derive the priority of the class used in packet scheduling.

B. TIME SYNCHRONIZATION

To achieve the required precision within the SUT, we need accurately synchronized clocks in the network. Therefore, in this section we outline standards offering time synchronization and available implementations.

TABLE 1. SR class requirements [10], [11], [12], [13] over 7 network hops.

SR Class	PCP	Max Latency	Max Jitter
A	3	2 ms	125 µs
B	2	10 ms	1000 µs
C (Audio)	5	15 ms	500 µs
D (Audio)	5	15 ms	500 µs
BE	0	no upper bound	no upper bound

IEEE 1588 [14] standard describes Precision Time Protocol (PTP) for precise time synchronization in a networked system. The individual clocks are synchronized via PTP instances running on each participating device. The participating devices are structured in a master-slave hierarchy. Master and slave exchange messages over the network and the slave clock is synchronized to the master clock. The reference time for the whole system is determined by the Grandmaster Clock (GM) clock, that sits on top of the hierarchy.

There are three types of PTP clock devices: Ordinary Clock (OC), Boundary Clock (BC), and Transparent Clock (TRCL). The OC is the simplest PTP device and can only run as a GM or a slave PTP instance. The BC has multiple BC ports, where each port behaves as the OC. The BC can become the GM, but then the node does not forward any PTP messages. In comparison to the OCs and BCs, the traffic controls (tcs) do not synchronize the clock of the machine they operate on. Instead, tc forwards the received protocol messages to the other ports and adjusts the time value in the PTP message according to the residence time in the tc. OCs and tcs may be combined.

On top of the PTP hierarchy is the GM. Any OC or BC can become a GM, so GM does not categorize as a dedicated clock type. The GM provides the time baseline within PTP network, ideally receiving its reference time through a reliable source, such as GPS. All other clocks are synchronized to the GM. Using the aforementioned clocks as building blocks of a PTP topology, we demonstrate how a complex PTP network is built. To establish the topology, a GM has to be chosen to dictate the timing baseline to which other clocks synchronize to. The Best Master Clock Algorithm (BMCA) determines the most suitable GM automatically by comparing various clock quality parameters.

PTP prunes cyclic paths in topologies, i.e., mesh topologies, to avoid cyclicity in the network [14]. Figure 1 illustrates this concept. We assume that the network connections between the BCs represent a mesh. Since there are two possible paths for synchronization of the bottom right BC, one of the paths is pruned to avoid cyclicity.

The PTP clock synchronization requires nodes to be exchanging timing information. The timing information is used to compute the clock offset and path delay between the nodes affecting the messages. Besides, the calculation of the message path delay between two nodes can be achieved in two distinct delay calculation modes, i.e., End-to-End (E2E) or Peer-to-Peer (P2P).

Figure 2 shows how a master and a slave clock exchange timestamps in the E2E mode. Equation 1 shows how the

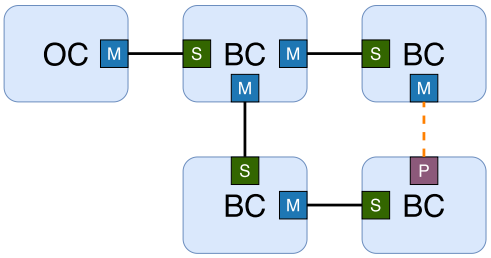


FIGURE 1. PTP topology with five nodes [14]. The dashed-line path is pruned, yielding a tree-shaped topology.

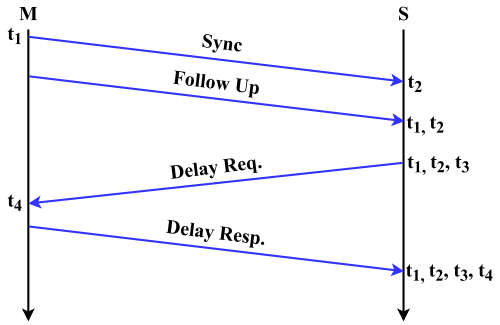


FIGURE 2. Sequence diagram of the two-step E2E synchronization model [14]. The slave keeps track of t_1 through t_4.

slave can compute the path delay d_P with the help of four collected timestamps. With the path delay, the slave clock can calculate the offset and accurately synchronize itself. An assumption is a duplex symmetric path between master and slave, and if it does not hold, the clock synchronization accuracy suffers [15].

$$d_P = \frac{(t_2 - t_1) + (t_4 - t_3)}{2} \tag{1}$$

The P2P mode calculates the delay in the other direction. Instead of using the Delay Req. and Delay Resp. message types, two PTP ports on different nodes enter a peer-to-peer relationship. They periodically exchange Pdelay messages to determine the delay between each other, which is then used in the clock synchronization procedure.

IEEE 802.1AS [9] standard relies on the methods defined in IEEE 1588 and applies them to the TSN in the form of a generic Precision Time Protocol (gPTP). In comparison to PTP, gPTP exchanges messages only at Layer 2 (using IEEE 802.1 MAC). Only two types of PTP devices exist in gPTP: PTP End Instances and PTP Relay Instances. The end instance corresponds to a PTP OC, whereas the relay instance is equivalent to a tc. The gPTP network is constrained, as packets are exchanged only between PTP instances and the gPTP clocks must operate on the same frequency. Overall, non-PTP devices cannot be used to forward PTP packets.

Within this work, we rely on the *linuxptp* project [16] to synchronize various clocks in the network. The project implements three individual tools - *ptp4l*, *phc2sys*, and *pmc*.

The *ptp4l* implements the PTP standard IEEE 1588 [14]. It can use Ethernet, IPv4, or IPv6 as an underlying protocol. To achieve nanosecond precision, it is crucial to use a Network interface card (NIC) supporting IEEE 802.1AS standard enabling hardware timestamping. Therefore, our setup uses COTS Intel® I210 NICs, which support the aforementioned feature. The *ptp4l* daemon runs on all interfaces, selects the most suitable GM, and synchronizes their clocks.

In case we need to synchronize additional clocks in the system, we rely on the *phc2sys* [17]. The *phc2sys* can run automatic mode in which *phc2sys* uses the information of *ptp4l* to synchronize clocks. This is important as the Intel® I210 NIC has as many Physical Hardware Clocks (PHCs) as the available ports. The tool is also needed synchronize the system clock to the GM.

C. LINUX NETWORKING STACK & TSN STANDARDS

First, we introduce how the Linux networking stack works and provide details on its implementation of individual TSN standards. The introduced TSN standards are relevant for the operation of IVNs and other TSN domains.

The networking process in Linux starts from the user space. As a first step, the application creates data (or a packet) that is passed using a system call into the kernel space. For each packet passed via a system call, two data structures are created: a buffer storing packet data, and a Socket Buffer (SKB) storing all of the packet's metadata. The packet with its corresponding data structures then proceeds down the ISO/OSI stack within the kernel space. It traverses through transport, network, and link layers. Each traversed layer adds the appropriate headers for the used protocols. Examples of the protocols which add their headers to the packet are: User Datagram Protocol (UDP) for Layer 4, IPv4 for Layer 3, and MAC for Layer 2. These steps are shown in Figure 3.

The TSN standards as defined in the IEEE 802.1Q [13] are meant for operation on the Link Layer. Within Linux, these are implemented as a part of the qdiscs. qdiscs are intermediary queues into which packets are inserted when the kernel intends to forward packets towards the NIC. Generally, there are two types of qdiscs: classless and classful. Classless qdiscs do not follow any hierarchy and can be considered as simple queues. An example of such qdisc is a Packet Limited First In, First Out queue (PFIFO), which is a straightforward first-in-first-out queue without any additional logic. Classful qdiscs are structured in a parent-child hierarchy. A parent qdisc can have several child qdiscs. Parent qdiscs generally contain logic determining into which child qdisc the packet is passed on. Only the child qdiscs at the end of the hierarchy has queues into which packets are enqueued.

The packet's priority is defined when it arrives in the kernel space and the corresponding metadata is stored in the accompanying SKB data structure. These priorities correspond to Traffic Classs (TCLs) encompassing packet's dedicated forwarding resources [18]. In essence, the TCLs are mapped onto one or more Tx queues of a given NIC. The Tx queue with the lowest number, and subsequently the highest priority, is emptied first. The qdisc itself is responsible for the pacing of the packets. It controls when a packet is forwarded to the software queue associated with the NIC, in Linux referred

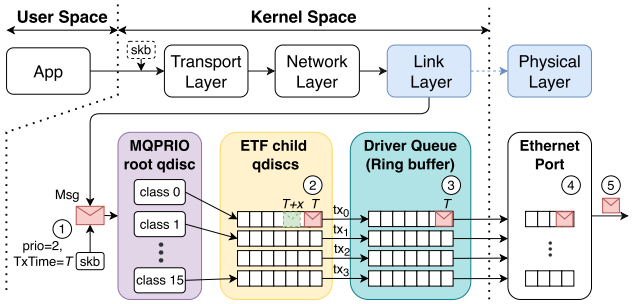


FIGURE 3. Overview of the packet going through networking stack for MQRPIO parent qdisc and ETF child qdisc. Encircled numbers correspond to explanation steps for ETF qdisc.

to as a ring buffer. This driver queue is read by the NIC, with packets then being stored within its hardware queues. Finally, the frames are processed by the NIC and transmitted onto the wire.

In practical applications, the command line tool *tc* is used to manage the qdiscs configuration in the networking stack [19]. For example, the Intel® I210 NIC has four hardware queues. Each of these queues can have a different child qdisc associated with it. The highest traffic priority corresponds to Hardware (HW) queue where the first qdisc is configured. As mentioned, the priorities stored in the SKB are read from their Virtual LAN (VLAN) PCP header field. This approach is described in [20]. For TSN, within the scope of this paper and the IEEE P802.1DG standard, we focus on Multiqueue Priority Qdisc (MQPRIO) and Time Aware Priority Shaper (TAPRIO) as parents and Earliest Time First (ETF) and Credit-Based Shaper (CBS) as child qdiscs. Some NICs support the respective standards in HW, allowing for their offload and possibly resulting in a speedup of the processing. In the following, we introduce the relevant TSN standards and their Linux configuration in detail.

1) EARLIEST TxTime FIRST

Figure 3 shows the functionality of ETF together with MQPRIO configured as a parent qdisc. As mentioned, during the packet lifetime a SKB storing packet's metadata is allocated. In ①, we see the priority and the TxTime. The TxTime is specified in the SKB SO_TXTIME option. Based on the priority value a packet is mapped to the corresponding class and later to a corresponding qdisc and HW queue. In ② we see, that that packets are sorted according to the TxTime T, where $T + x > T$. Next ③, the packet is handed over to the ring buffer `delta` ns before the TxTime from which the packet is read to the Ethernet port on the NIC. The NIC stores the frames in its own queue ④, and sends the packet at the TxTime according to its hardware clock ⑤. Of note, we do not have visibility on the packet once it leaves the ring buffer until it reaches the next hop.

ETF can operate in two modes - strict and deadline. With the strict mode, the packets are dequeued at the TxTime. In case of the deadline mode, the packet can be dequeued anytime before the TxTime is reached. The expected outcome when using the deadline mode is a lower delay but a higher jitter. As mentioned, we need to account for a delay before the TxTime, when the ETF qdisc awakens and dequeues the packet towards the NIC. The corresponding parameter is the `delta` which also serves as a fudge factor for the system delay. With HW support of the NIC, the ETF operation can be offloaded for higher precision.

2) TIME AWARE PRIORITY SHAPER

IEEE 802.1Qbv [8], [21] known as Time-Aware Shaper (TAS), TAPRIO qdisc [22] in Linux, or "Enhancements for scheduled traffic" as a part of the IEEE 802.1Q-2018 [13] standard. It offers support for synchronized scheduling of multiple tcs on a single interface. The traffic flow is controlled by gates for each traffic class, that operate according to a cycle determined by the system configuration. The packets can be dequeued only when the gate is opened. The functionality is similar to Time Division Multiple Access (TDMA) scheme.

The various TCLs have a dedicated transmission window within a cycle of configurable length. The gate of the given class is open during its window. The packets are passed to the child qdisc only if enough time remains for the transmission until the gate closes. To ensure that the individual window cycles and windows do not interfere, we can add guard windows. These can have the size of a given packet serialization time computed using the packet size and link speed. In scenarios when the transmission duration of a frame is unknown in advance, e.g., in cut-through switching, the length of the window should be compensated with an appropriate schedule configuration.

Linux TAPRIO qdisc configuration maps the tcs to HW queues and their corresponding transmission windows using their assigned priorities. TAPRIO is enabled by the child ETF qdisc, which is configured for each HW queue of a corresponding port. ETF offers control over the sending time of its packets. Due to that, some literature refers to it as "LaunchTime" feature. For sorting the packets in queues, their pre-defined transmission time is used. The packets are kept in the queue by the qdisc until this deadline arrives.

Figure 4 shows a configuration with eight traffic classes and gates. TAPRIO must be configured as a parent qdisc to manage the mappings of the PCP to TCLs with respect to the gate opening. In this case, only a single gate is opened to allow the traffic dequeuing. In addition, the frame selected for transmission can also depend on the used child qdisc. As an example, CBS could be used as the child qdisc, restricting the sending rate despite the gate being open by TAPRIO.

TAPRIO is part of the synchronous TSN standards. Therefore, it requires to align the schedules of all devices in the network. This is achieved by setting the `base-time` properly. The parameter indicates a start time of the schedule and is set in nanoseconds.

To configure the duration for which the gate is opened, the `sched-entry S $MASK $DURATION` parameter is used. `$DURATION` is the time window duration, and `$MASK`

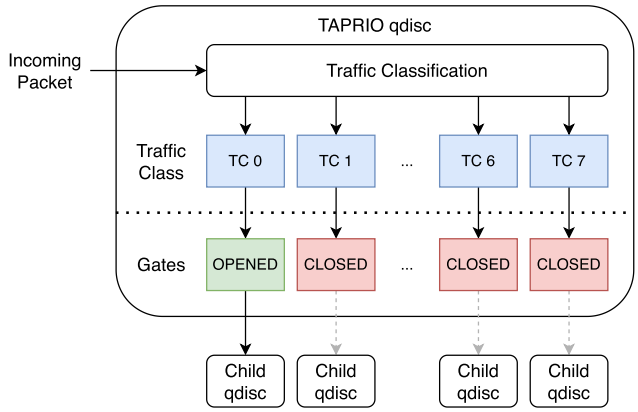


FIGURE 4. Overview of TAPRIO qdisc with tcs mapped to HW queues and corresponding gates [8].

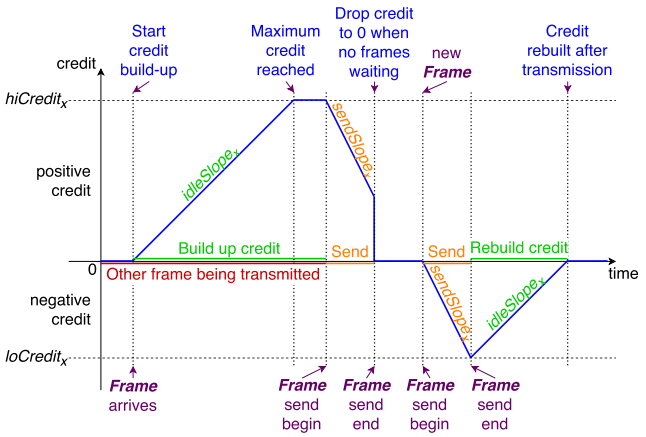


FIGURE 5. Example credit level over time during CBS algorithm operation [13].

indicates which gate is opened/closed during this window. Sum of all the `sched-entry` is the cycle time.

Additional configuration parameter is `flags`. It supports the TxTime mode (`flags 0x1`) and the full offload mode (`flags 0x2`). The TxTime mode automatically sets the packet's TxTime, which is important for applications that do not set this time. ETF qdisc uses the TxTime to control when a packet is sent and in case no TxTime is set or the packet arrives to the queue after the TxTime passes, the packet is dropped. When the `flags 0x1` is used we need to configure also the `txtime-delay` that accounts for the system delay. As per documentation, it should be greater than the ETF qdisc `delta` value.

When combining the TAPRIO with ETF, we have to consider their corresponding parameters. The `txtime-delay` is added to the overall TxTime of a given packet. The packet is passed through the parent qdisc gate once its open for a given priority to the ETF queue. In case the windows are too large, the packet may reach the queue too short before the TxTime and is dropped. This is especially a problem if the `delta` value is set too high. Similarly, when the window sizes are too small, the `txtime-delay` has to account for that, as the packets might spend additional time waiting before the TxTime is reached.

3) CREDIT-BASED SHAPER

IEEE 802.1Qav [7], also known as the CBS algorithm belongs to the IEEE 802.1Q-2018 [13] family of standards. The algorithm is used to select frames that will be transmitted next on the interface from a set of SR classes and their associated hardware queues. CBS itself enables bandwidth allocation to pre-defined SR classes. The algorithm ensures this allocation for every SR class using a scheduling system based on credits and with that offers some bounds on delay, jitter, and packet loss.

The algorithms' credit-based operation is visualized in Figure 5. The start of a frame transmission is allowed only when the collected credit is ≥ 0 and no frames from other SR classes are currently being transmitted by the NIC. The amount of accumulated credit over time for any SR class X is governed by four parameters:

- $hiCredit_X$ - maximum allowed credits
- $loCredit_X$ - minimum, negative, allowed credits
- $idleSlope_X$ - rate at which credits are replenished
- $sendSlope_X$ - rate at which credits are spent

Credit is accumulated when at least one packet is present in the queue of class X. When a frame is present in the queue of class X and a frame from another SR class is being transmitted by the NIC, the credit is accumulated at $idleSlope_X$ rate until $hiCredit_X$ is reached. After the other frame's transmission is over, the frame of class X can be transmitted. During transmission, credit is spent at $sendSlope_X$ rate. The amount of accumulated credit can be negative but never lower than $loCredit_X$. If there are no additional frames in the queue of class X and the associated credit was > 0, the available credit is set to 0. When a new frame arrives for class X, and no frame from other SR class is being transmitted, the packet can be sent out immediately. In this case, the credit is spent at $sendSlope_X$ rate during transmission and it is replenished afterward at $idleSlope_X$ rate until it reaches 0 again.

The values of the four aforementioned parameters are calculated based on the bandwidth fraction B_X allocated to the SR class X, the maximum frame size in bits MFS_X for the class X, the NIC transmission rate PTR in bit/s, and the maximum frame size in bits expected on the associated NIC MFS_0. In the following equations, we denote any other SR classes using the interface as class Y. Of note, CBS configuration includes the Physical Layer (PHY) overhead in the frame size.

$$idleSlope_X = B_X \cdot PTR \tag{2}$$

$$sendSlope_X = idleSlope_X - PTR \tag{3}$$

$$loCredit_X = MFS_X \cdot \frac{sendSlope_X}{PTR} \tag{4}$$

$$hiCredit_X = idleSlope_X \cdot \left(\frac{MFS_0}{PTR} + IF_X\right) \tag{5}$$

$$IF_X = \sum_{Y}^{Y<X} \left(\frac{hiCredit_Y}{-sendSlope_Y} + \frac{MFS_Y}{PTR}\right) \tag{6}$$

TABLE 2. PCP to TCL mapping according to the IEEE 802.1Q specification [13].

PCP	0	1	2 (SR-B)	3 (SR-A)	4	5 (SR-C/D)	6	7
4-TCLs	0	0	2	3	1	1	1	1
8-TCLs	1	0	6	7	2	3	4	5

The $idleSlope_X$ parameter is defined in Equation (2). It specifies the rate at which a class accumulates tokens while frames are queued in their corresponding queue and directly corresponds to the bitrate allocated to an SR class X. The $sendSlope_X$ follows in Equation (3) as a difference of NIC's bitrate subtracted from the bitrate allocated to class X. $loCredit_X$ is specified in Equation (4), signifying that at least one packet of MFS_X size can be transmitted when credit is 0. The $hiCredit_X$ shown in Equation (5) is a combination of the rate which is guaranteed for class X and the maximum interference that can be caused by best-effort traffic or any other SR class with higher priority than X. The SR class priorities introduced in Table 2 show a default mapping of PCP to TCL. Higher TCL value corresponds to a higher priority within the packet scheduler. The table distinguishes between the number of available tc classes in the system and presents values for 4 and 8 classes corresponding to NICs with 4 and 8 queues respectively. Given that mappings, SR class A has the highest priority and SR class B the second-highest priority. The interference from other SR classes is introduced in Equation (6).

To mention, the parameters introduced above use B and kbit/s in Linux, but the IEEE 802.1Qav [13] standard defines them in bit or bit/s. As an example, for the Linux configuration with packets of 1250 B at a bitrate of 100 Mbit/s and maximum packet size of 1542 B with interface bitrate of 1 Gbit/s, the corresponding qdisc parameters for SR class A are: $idleSlope_A = 100000$, $sendSlope_A = -900000$, $hiCredit_A = 155$, and $loCredit_A = -1125$.

Furthermore, in Linux, CBS needs to be associated with MQPRIO qdisc to distinguish between packets corresponding to specific SR classes. The MQPRIO enables mapping of SR classes to defined priorities and hardware queues of the NIC and is an implementation of the strict priority forwarding [23]. The CBS qdisc are configured per HW queue as MQPRIO's child qdiscs.

D. THE EnGINE FRAMEWORK

An important addition to the introduced methodology is the infrastructure and framework for experiment evaluation called *EnGINE* (**En**vironment for **G**eneric **I**n-vehicular **N**etwork **E**xperiments) [3]. The framework uses Linux networking stack, which as mentioned offers a variety of qdiscs configurations together with and TSN capable COTS NICs. *EnGINE* supports IEEE 802.1Qav [7], IEEE 802.1Qbv [8], and IEEE 802.1AS [9] standards introduced previously and their corresponding qdiscs along with the ETF and MQPRIO qdiscs. The framework is designed for TSN experiments with a focus on smart manufacturing and IVNs. Therefore,

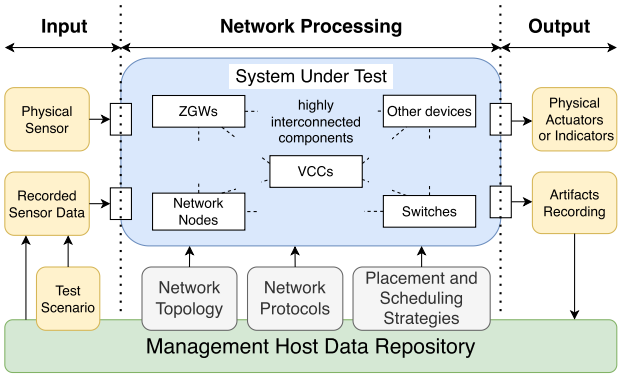


FIGURE 6. Overview of experiment components [3].

it fulfills the requirements introduced in Section II-A with additional details covered in [3].

The framework orchestration tool is built using *Ansible*,[1] which brings flexibility to the network and data sources configuration. It allows for traffic generation or replay, collection of artifacts for further evaluation, and post-processing of results providing valuable insights. As a part of the requirements, the experiments are reproducible and configurable.

Figure 6 shows the structure of an experiment within the *EnGINE* framework. It consists of three elements - **input**, **SUT**, and **output**. First, **input** defines a scenario and the traffic type under which the network is tested. Second, **SUT** contains the networked elements used in an experiment. *EnGINE* allows the network structure to be configured for various topologies and network configurations. Finally, **output** handles the collected experiment results.

In practice, each experiment campaign has four phases: **install**, **setup**, **scenario**, and **process**. During the **install** phase, the required nodes are booted with the Operating System (OS) image of preference. This step utilizes the plain orchestration service (*pos*) [24]. The second, **setup** step ensures the required software artifacts are properly installed and prepared for the individual experiments. The third, **scenario** handles the execution of the individual experiments. During this, a network topology, individual interfaces and also *linuxptp* for clock synchronization are configured for each experiment. Next, all used applications during the individual experiments are started. As an example, an application could be a traffic generator or packet capture. If needed, we can also add dynamic behavior to the experiment, e.g., switching off a link or introducing an additional traffic path. The generated artifacts of each experiment are **processed** on the individual node, and on the management host to which all of the artifacts are copied to. To note, post-processing is possible on individual or several experiments at once.

III. RELATED WORK

Several resources describe application domains of TSN providing understanding of common requirements and challenges. For our methodology, we rely on various traffic 6 and

[1] https://www.ansible.com, Accessed 15.07.22

sources identified via a brief survey of TSN traffic characteristics. Finally, we provide an overview of other approaches used in building of testbed infrastructure focusing on TSN experiments and measurements.

A. APPLICATION DOMAINS OF TSN

The survey publications [2] and [1] investigate the current state of technology and ongoing developments in the field of IVNs. The ongoing focus on ADAS and multi-media functions in the vehicles results in an increased transferred data volume. Network gateways connect sensors and actuators partially using other bus systems such as Controller Area Network (CAN), Local Interconnect Network (LIN), or FlexRay [1], [2]. Ethernet with TSN, due to their low cost, flexibility, and popularity have the capability of becoming the backbone of the IVN.

In [25] the authors present a case study of an aerospace application using TSN. The publication provides an overview of possible traffic flows and analyses the network and hardware bottlenecks in such domain. Even though, not all of the TSN standards might be recommended or used, a subset of them is considered by the authors. TSN support for satellite development, new recommendations, and Quality of Service (QoS) guarantees for networks inside satellites are discussed in [26]. The authors analyze the state-of-the-art standards and architectures while providing recommendations on transition to Ethernet networks with TSN. For that, they summarize the available traffic patterns and their requirements with respect to bounded latency, jitter, and data rate.

Within [27] the authors focus on the aspects of OPC Unified Architecture (OPC UA) along with TSN and its applicability to industrial automation. They design an architecture specifying how industrial automation with Internet of Things (IoT) applications can benefit from TSN. The authors separate the network functionality on Layer 2, which is crucial for TSN, from Layer 3 for cloud connectivity. TSN offers bounded latency, jitter, and a very low-packet loss ratio suitable for such deployments.

Audio Video Bridging (AVB) that closely relates to TSN also offers a new set of use-cases and scenarios. In [28], a theoretical evaluation of AVB and TSN protocols within the IVNs is performed. The authors use CBS, TAS, and SR classes categorization similarly as we introduce in this work. In addition, as analyzed in [29], AVB considers Frame Preemption IEEE 802.1Qbu together with Cyclic Queuing and Forwarding IEEE 802.1Qch [28]. The authors also compare AVB scheduling with the IEEE 802.1Qbv standard [21]. Finally, the publication summarizes other standards in the TSN domain and mentions open issues with them.

B. TRAFFIC CHARACTERISTICS OF TIME SENSITIVE NETWORKS

Several publications look into the characterization of the traffic within IVNs. The open dataset ApolloScape [30] gives an impression of the high throughput required to transport sensor data over the network. In [31], the authors describe traffic and topologies of IVNs. Furthermore, they evaluate the impact of traffic shaping on the various traffic classes. A detailed overview of common traffic types in industrial automation systems and recommendations for suitable traffic shaping is presented in [32].

C. TSN TESTBEDS

A poster publication [33] describes a TSN testbed managed by a central Software Defined Networking (SDN) controller. Similarly to *EnGINE* framework, only COTS hardware is used. Unfortunately, an in-depth description of the approach and its evaluation is not available.

Another report describes a testbed with TSN capabilities using custom FPGA-based hardware extensions [34]. It is used to assess metrics such as latency and jitter of the TAS. An important conclusion is that end nodes software stacks are the main contributors to outliers in the packet delays.

[35] studies the integration of the industrial communication protocol OPC UA PubSub into TSN. The OPC UA software stack has been improved to achieve low and deterministic latency. In contrast to our approach presented within this work, the measurements the authors performed used point-to-point node topology with a fixed traffic pattern.

In another study, a TSN testbed was built to assess the functionality of a simulation framework [36]. The topology within the experiments is fixed, consisting of two proprietary TSN hardware switches, four talkers, and four listeners. The authors show that a good similarity between the results from simulation and physical testbed could be achieved.

Finally, the Industrial Internet Consortium (IIC) operates two TSN testbeds on two continents [37], [38]. Several TSN manufacturers integrate their solutions and test for device compatibility to show TSN support. A main goal of the testbed is to achieve interoperability by standard compliance.

D. MODELLING OF TSN

In [39], Network Calculus was used to assess the performance characteristics of the various TSN traffic shapers and also possible combinations thereof. The publication furthermore contains guidelines for flow shaping under certain requirement trade-offs.

Instead of using TSN traffic shaping to achieve bounded latencies in IVNs, a more simple approach of using FIFO queues without reshaping at the switches was studied in [40]. Additionally, a complete framework for distributed time-sensitive embedded applications communicating over TSN is presented in [41]. The authors combine a timing model of the software stack and of the network stack.

E. SIMULATION OF TSN

Several, both open-source and proprietary, TSN simulators are currently available. Core4Inet [42] and NeSTiNg [43] frameworks, together with recently extended INET

framework[2] [44], bring TSN capabilities to the open-source discrete event simulator OMNeT++ [45]. For example, Core4Inet is used in the evaluation of a heuristic-based TSN scheduling algorithm to compute gate control lists for the TAS [46]. Furthermore, the INET framework stresses its capabilities supporting a realistic TSN deployment as well as the framework's applicability to IVN simulation.[3]

Another study focuses on the assessment of the CBS with realistic automotive network traffic using the OMNeT++ simulator [47]. The authors indicate that bounded jitter, an important promise of the CBS algorithm, can be achieved in this scenario. Another proprietary solution is the Real-Time at Work (RTaW) Pegase software[4] which is used as an evaluation tool in some publications such as [25] and [31].

IV. THE METHODOLOGY & CASE STUDY

This section analyses the introduced methodology and covers a sample use-case relevant for the IVNs which we consider as a case study of the introduced methodology.

A. METHODOLOGY INTRODUCTION

This work has two fundamental building blocks, as briefly outlined in Section I. The first is the infrastructure framework *EnGINE*. The second is the methodology with individual steps and actions showing how to utilize the infrastructure to collect valuable insights for suitable TSN configuration and system optimization to achieve defined KPIs.

The *EnGINE* framework [3] offers the infrastructure for reproducible TSN experiments ranging from small deployments (two nodes and a single traffic flow) to more complex scenarios (up to thirteen nodes and numerous flows). The framework was designed with flexibility and scalability in mind to allow for the evaluation of various topologies that can represent realistic IVNs and encompass current and future use-cases that can be encountered in such scenarios.

The methodology steps investigate the TSN standards and their applicability in various domains. For **S1**, we need to define requirements and traffic patterns present in a given (TSN) domain. **S2** focuses on configuration and evaluation of TSN standards using the COTS hardware and open-source solutions, but not limited to such solutions. Lastly, the **S3** looks into system optimization needed to meet defined KPIs. In the following paragraphs, we focus mostly on the example of IVNs and outline how the *EnGINE* framework is used to follow the individual steps.

1) **S1** - SYSTEM REQUIREMENTS

Starting with the **S1**, we provide an overview of the requirements present in IVNs. A similar investigation can be performed for any other domain. Section II-A defines requirements on the methodology itself with respect to the TSN experiments within the scope of IVNs. An important factor is the analysis of possible traffic patterns within the IVNs and their categorization to SR classes and their mappings to the tc. Table 1 indicates what bounded latency and jitter each SR can tolerate over seven hops and to which PCP priority the traffic should be assigned. To note, a different traffic pattern might require a different qdisc configuration to achieve the required properties and system performance characteristics. For instance, if CBS qdisc is not configured correctly for a specific traffic pattern, we might observe large delays due to the time required to build up credit and possible packet drops due to overflowing queues.

A factor that is not considered within the SR classes is the allowed packet loss. Based on literature identified in Section III, we assume that packet loss might vary among different real deployments and applications using the given data. Nevertheless, we aim to offer as low packet loss as possible, which we achieve with the proper configuration of given parameters and optimization of the OS.

A crucial aspect in the IVNs is redundancy to overcome failures, i.e., link, interface, or a node failures. Therefore, the deployed topologies offer multiple connections among the Zonal Gateways (ZGWs) or Vehicular Control Computers (VCCs). We consider such mesh-like topologies in our configuration, but the protocols investigated within the scope of this work do not offer additional redundancy. Of note, within the TSN, IEEE 802.1CB considers Frame Replication and Elimination for Reliability (FRER) that duplicates every packet between source and destination [48]. Nevertheless, evaluating this solution is out of scope for our approach, as we focus mainly on fulfilling the SR class requirements.

Finally, COTS HW and custom Linux kernel cannot offer real-time guarantees. Therefore, we identify suitable ways to optimize the Linux OS to fulfill the requirements as described in Section V and supported by the definition of **S3**.

2) **S1** - ANALYSIS OF AVAILABLE TRAFFIC PATTERNS FOR IVNs

The second part of **S1** focuses on the definition of traffic patterns and is the first step when defining the domain. As TSN aims to be used within real-time environments that usually have a limited scope, identifying and analyzing a list of present traffic patterns is possible. Table 3 summarizes the individual traffic patterns generally found in IVNs based on literature and our internal analysis. The table not show an exhaustive list of all possible traffic patterns and types, but should provide enough insights to apply the methodology to other domains or use-cases. The list provides various categories from sensors needed for the ADAS covering video, Light Detection and Ranging (LiDAR), radar, and ultrasound, entertainment systems for audio, video, and file transfers, diagnostics, and Command & Control (C&C) traffic for network management. Besides, the table categorizes whether the traffic is periodic, to which SR class it belongs, including a corresponding PCP priority, and summarizes the frame sizes,

[2]https://inet.omnetpp.org/2022-06-15-tsn-released.html, Accessed 17.07.22

[3]https://inet.omnetpp.org/docs/showcases/tsn/combiningfeatures/invehicle/doc/index.html, Accessed 17.07.22

[4]https://www.realtimeatwork.com/rtaw-pegase/, Accessed 17.07.22)

inter-frame packet spacing, and generated throughput for a given family of sensors.

To fill in the aforementioned table with the traffic patterns, we started with resources presented in Section III-B. In [31], the authors provide a case study of various traffic patterns that are implemented and evaluated in the RTaW simulation tool. It provides an overview of the number of streams, packet sizes or their range, timing constraints, and categorization. To note, for all traffic patterns we selected the upper part of the range for the frame sizes.

A similar notion of traffic patterns is introduced in [2], providing categorization of different traffic classes with respect to their latency. In [1], the authors do not only focus on Ethernet-based solutions, but also on other bus systems, e.g., CAN and LIN, and provide frame parameters information for them. The publication [49] compares a TSN network with a cut-through switching optical architecture. The solution is proposed using OMNeT++ simulator with Core4INET framework. For their analysis, they describe various traffic patterns, e.g., ADAS video, infotainment, and other sensors, used for their evaluation. [40] evaluates TSN for the use-case of IVNs and provides assumptions on the used traffic within their theoretical evaluation model.

We also utilize the SR class recommendations for audio with strictly defined period, frame size, and priority [10], [11], [12]. These are used to define the corresponding SR classes and PCP. Next, we derive our traffic patterns based on available physical sensors within *EnGINE* framework, such as cameras and LiDAR, or derived traffic patterns from publicly available documentation. Having access to the physical device allows for detailed traffic pattern analysis, possible modification of parameters, and recording of the traffic to packet captures. It is especially relevant for aperiodic traffic patterns, as we can observe traffic bursts from the packet capture. However, using own hardware has two main drawbacks - the need of the device and packet format differences among manufacturers resulting in various traffic patterns.

As a result, we also evaluate other available data sources, e.g., the Apollo dataset [30], Waymo open-source dataset [50], and the Oxford Robotcar dataset [51]. All of these datasets provide quality footage from a plethora of sensors used for autonomous driving. Unfortunately, none of them offers the data in a suitable format, e.g., packet capture, that could be later on replayed in the network for additional analysis. It must be noted that these datasets are mainly used for the training of machine learning models.

3) **S2** & **S3** - SYSTEM CONFIGURATION

The **S2** focuses on providing required steps to properly configure various qdisc parameters relevant to the corresponding traffic pattern (e.g., CBS `idleSlope` parameter and window sizes for TAPRIO) or which are system dependent (ETF delta value). The underlying infrastructure - the *EnGINE* framework - allows to evaluate the suitable configuration and assess if the requirements can be fulfilled. The details of the **S2** parts are described in Section V where we focus on the setup and configuration aspects and Section VII that provides the evaluation details, based on the defined experiments from Section VI. On the other hand, the **S3** focuses on the preliminary results, that do not fulfill the defined requirements and showcase various artifacts caused by the operating systems. Section VII covers in detail possible optimization for TAPRIO with respect to traffic patterns, window sizes, and window alignments. The goal is to identify the cause of the artifacts and optimize the system to mitigate them. Section V offers a sample configuration that showcases the observed artifacts and ways how to remove them.

4) SUMMARY OF THE METHODOLOGY

The infrastructure (*EnGINE* framework) and the methodology steps Sections IV-A1 to IV-A3 form the building blocks needed to optimize the system performance towards defined requirements in a form of KPIs. A possible challenge is the mapping the observed traffic patterns and network loads within a given domain. The challenge is especially present in large-scale systems, but real-time systems are more constrained and identifying high-priority traffic should be feasible. Once the traffic is identified, we can devise a set of experiments to evaluate suitable configurations and evaluate the requirement fulfillment. Finally, in case the KPIs are not met, we have to optimize the system to constrain scheduling overhead or context switching.

5) APPLICABILITY TO OTHER TSN DOMAINS

Following the methodology steps, we believe it is not limited to the IVNs and can be generalized and used in other TSN deployments. The related work supports this in Section III-A, showing that the TSN is being actively researched in other domains. First, industrial manufacturing and OPC UA protocol aim to standardize the profile IEC/IEEE 60802 for Industrial Automation [6]. Nevertheless, TSN is not the only part of the architecture, as for certain scenarios Layer 3 functionality is also important. Next, the audio-video bridging is not only heavily discussed within the vehicular domain, but also by audio-video equipment manufacturers [52]. Lastly introduced is the IEEE 802.1DP profile - TSN for Aerospace Onboard Ethernet Communications will be standardized soon [5], which proposes solutions for aerospace, covering airplanes, helicopters, and satellites [26].

The methodology originates from the vehicular domain, but after a detailed analysis of other TSN domains, we see many similarities with respect to the requirements, used TSN standards, and layers of operations, which is summarized in Table 4. The requirements focus on the QoS with respect to bounded latency, jitter, low packet loss, and reliability. The list of TSN standards is not exhaustive but provides an overview of commonly used standards. Based on the related work and profiles, the selected standards are dependent on the specific use-case within a given domain. Similarly, many of the other domains focus on Layer 2 functionality, but possible extensions to Layer 3 could help if the underlying layer cannot fulfill the requirements. Finally, all of the domains

TABLE 3. Overview of various traffic patterns present in IVNs and their required throughput.

Flow	Description	Streams per Flow	Aperiodic	SR Class	PCP (4TCs)	Period [µs]	Frame Size [B]	Packets per Second	Throughput per Flow [Mbyte/s]	Total for all Flows [Mbyte/s]	References
0	C&C*	6	NO	A	3	1000	1024	1000	1.024	6.144	[31], [49]
1	C&C*	6	NO	A	3	6000	1024	167	0.167	1.026	[31], [49]
2	Vid. ADAS†	2	NO/YES	A/B	3/2	600	1500	1667	2.501	5.001	[31]
3	Vid. ADAS†	6	NO/YES	A/B	3/2	400	1500	2500	3.750	22.500	[31], [49]
4	Audio	8	NO/YES	B/C/D	5	10000	256	100	0.026	0.205	[31]
5	Lidar	3	NO/YES	A/B	3/2	2500	1500	400	1.350	4.050	Internal
6	Radar	6	NO/YES	A/B	3/2	2500	1250	400	0.500	3.000	Internal
7	Ultra sound	3	NO/YES	B/BE	2/0	20000	512	50	0.026	0.077	Internal
8	GPS	1	NO/YES	A/B	0	10000	256	10	0.003	0.003	Internal
9	Audio SR C	3	NO	C	5	1451	256	690	0.177	0.530	[10]–[12]
10	Audio SR D	3	NO	D	5	1333	256	751	0.192	0.577	[10]–[12]
11	BE‡	4	YES	BE	0	200	64	5000	0.320	1.280	[31]
12	FT¶	5	YES	BE	0	200	64	5000	0.320	1.600	[31]
13	Diagnostics	5	YES	BE	0	200	64	5000	0.320	1.600	[31], [49]
14	BE‡/FT¶	3	YES	BE	0	200	512	5000	2.560	7.680	Internal
15	BE‡/FT¶	3	YES	BE	0	200	1500	5000	7.500	22.500	Internal

*Command&Control; †Advanced Driver-Assistance System; ‡Best Effort; ¶File Transfer

TABLE 4. Overview of TSN domains with respect to the methodology.

TSN Domain	Req.†	TSN Standards†	Layers	References
IVN		IEEE 802.1Qav,	L2	[4]
Aerospace	BL, BJ,	IEEE 802.1Qbv,	L2	[5]
AVB	LPL, R	IEEE AS-Rev,	L2	[10]–[13]
Manufacturing		...	L2/L3	[6]

†Applies to all TSN domains
BL - bounded latency; BJ - bounded jitter; LPL - low packet loss;
R - reliability; Req. - requirements

have rather bounded network sizes with respect to traffic patterns, traffic criticality, and the scale of nodes.

Therefore, using the described steps and analysis of other deployments, we believe the methodology is applicable to other domains. In case additional TSN standards need to be evaluated, the underlying *EnGINE* framework can be extended to cope with additional standards to assess the qdiscs configurations and evaluate their performance.

B. EXEMPLARY USE-CASE

The **S1** details the available traffic patterns presented in the domain of IVNs. A subset of the traffic patterns is used within the exemplary use-case. Figure 7 shows the overall infrastructure offered by the framework along with a sample topology emulating a high-end vehicle with six ZGWs and one VCC. The high-end topology is used as a base for our exemplary use-case, which is later extended in Section VI. The use-case covered in detail comprises of five connections with varying or same number of hops, sample traffic patterns for different PCP traffic classes, and cross-traffic on a subset of the nodes. Depending on the configuration, different nodes act as a source or a sink. With respect to **S2**, Section V provides details on the used tools and their precision along with OS optimizations and configuration parameters. Detailed design of experiments is shown in Section VI, offering steps on how to configure the qdiscs based on the provided requirements. Finally, we provide results in the Section VII for the experiment campaigns that showcase the fulfillment of the requirements and verify the methodology capabilities.

The exemplary use-case serves two purposes. First, it verifies the newly introduced methodology following the individual steps - definition of requirements and traffic patterns, the configuration of the qdiscs, and optimizations to fulfill the given KPIs. The use-case is from the domain of IVNs, but once suitable traffic patterns are identified, e.g., from the domain of aerospace, the qdiscs can be configured accordingly and evaluated with respect to their requirements. The resulting configuration of the experiments associated with the use-case is outlined in Section VII-B.

Second, the use-case provides and guides through the details of the individual parts of the infrastructure capabilities and configuration aspects. Section V introduces the performance evaluation of used tools, observed artifacts in Linux without OS optimization techniques, and system precision. Furthermore, it explains in more detail the challenges of qdisc configuration. This is a fundamental aspect of getting an understanding of observed behavior and identifying possible limitations. Next, Section VI provides the relation with the traffic patterns and their applicability to the given use-case. The Section VII shows additional results focusing on the evaluation of performance up to seven hops. Even though, the sample use-case does not include many hops, it serves as a format that can be built and expended on. As seen in Figure 7, the infrastructure supports more than seven hops between source and sink. The HW specification can run numerous flows, various traffic patterns generated live or replayed from recorded packet captures, and network topologies matching real-world scenarios.

V. EXPERIMENT SETUP

In the following, we introduce the experimental setup, including the infrastructure provided by the *EnGINE* framework. We furthermore showcase the utilized traffic generation applications, as well as outline the system precision and ways

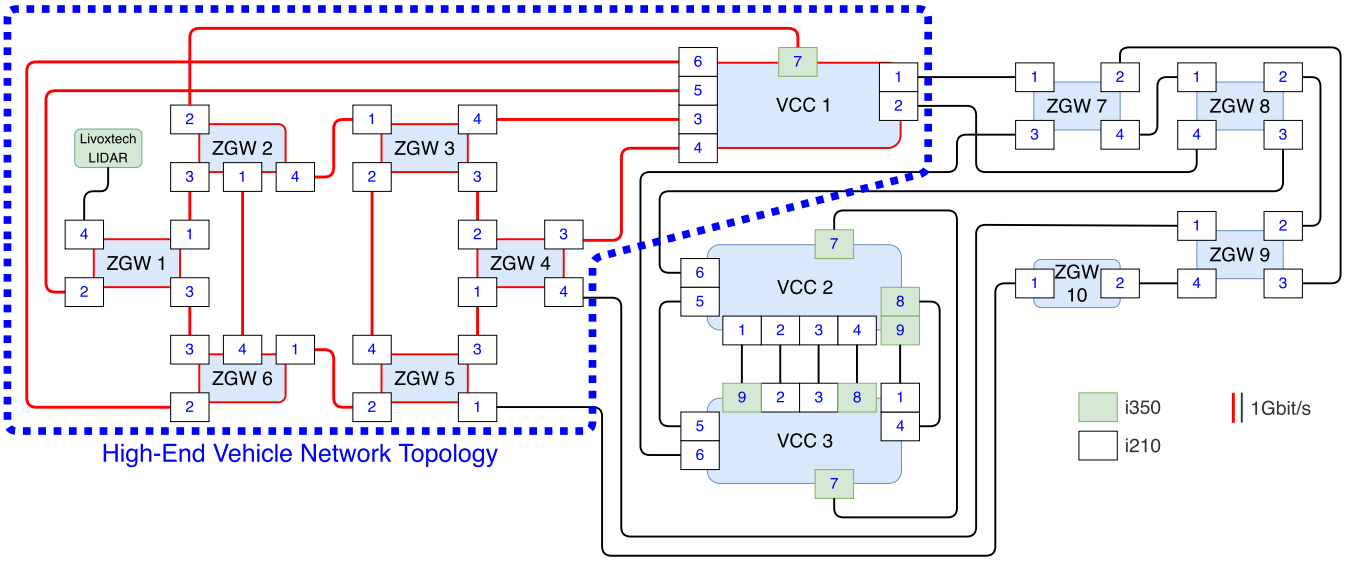


FIGURE 7. Simplified *EnGINE* infrastructure overview. Red outline of the nodes and links indicates elements used for high-end in-vehicular network topology.

TABLE 5. Hardware used for experiments with details on supported TSN standards by NICs [3].

	High-Performance VCC	Low-Performance ZGW
CPU	4C/8T Intel® Xeon D-1518	4C/8T Intel® Xeon E3-1265L V2
RAM	128 GB of DDR4 Memory	16 GB of DDR3 Memory
NIC	6 × 1 GbE Intel® I210†	4 × 1 GbE Intel® I210†
	4 × 1 GbE Intel® I350‡	1 × 2.5 GbE Intel® I225†
	2 × 10 GbE Intel® X552‡	2 × 10 GbE Cisco Nexus GM‡

†IEEE 802.1Qav, Qbv, AS; ‡IEEE 802.1AS

in how to optimize it. Finally, we go into detail on how to configure the traffic shaping in the network upon consideration of available network topologies. These aspects are relevant for the upcoming experiment design and evaluation. First, we provide details on the tools used to affect the evaluation results. Next, we optimize the system to mitigate certain artifacts caused by kernel resource management. Finally, we evaluate system dependant qdisc parameters that have to be taken into consideration. The last steps contribute to the **S3** step of the methodology with respect to system optimization.

Table 5 shows an overview of available and used hardware for experiments and what TSN standards the NICs support. This is a representation of devices we performed our experiments with and a showcase indicating what hardware is available within the *EnGINE* framework deployment. Mostly, we use the Intel® I210, I350, but for other experiments, we could also rely on the I225 and X552 NICs which are not investigated further within this work. To note, it is possible to use any NIC that does not support any standards and purely relies on the SW support provided by qdiscs.

A. TRAFFIC GENERATION

In our experiments, we mostly rely on synthetic traffic generators to produce traffic patterns outlined in Table 3. The following applications can be generally used to create periodic traffic. With some modifications or custom configurations, these could also be used to generate certain aperiodic traffic patterns, however, we do not focus on those within the scope of this publication. As such traffic patterns are challenging to generate, we omitted them but see a potential for their emulation using realistic devices, e.g., LiDARs or cameras. For some of the applications, we additionally provide configuration insights to achieve the desired traffic. Finally, we perform tests using certain applications, verifying their accuracy in traffic generation and applicability to the methodology introduced within this work.

1) Iperf3

For generation of synthetic, periodic traffic patterns we utilize the `Iperf3`[5] network performance measurement tool. The tool's primary use is to evaluate the achievable network throughput between two hosts in a network. To achieve that, `Iperf3` generates a stream of packets towards the other end-point that aims to saturate the link, in essence transmitting as many packets as possible through the network. Thanks to its flexibility, it also allows for creation and generation of periodic traffic patterns between a source and a sink. These patterns can be configured using the `-b` and `-l` options of the `Iperf3` client. The aforementioned parameters enable setting of a specific target bitrate R_{I3} in bit/s and payload size B_P in Bytes respectively. Since the required traffic is usually specified as a combination of PHY frame size B_{PHY} and period t_P, we derive Equation (7) and (8) allowing us to adequately prepare the client. Of note, to find the B_P we also need to consider the overhead B_O induced by various layer headers, in order to achieve the desired B_{PHY} as shown in Equation (7). For instance, UDP traffic transmitted via a VLAN-tagged network, we derive $B_O = 70$ Byte as shown in

[5]https://iperf.fr/, Accessed 01.07.22

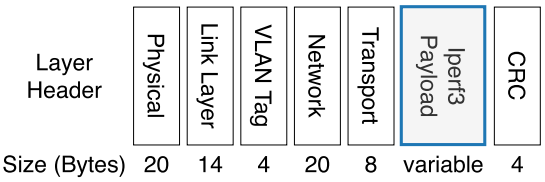


FIGURE 8. Structure of PHY `Iperf3` frame with UDP as transport layer.

Figure 8. The R_{I3} is defined as a product of `Iperf3` payload size in bits and the frequency with which they are supposed to be sent on the wire.

$$B_P = B_{PHY} - B_O \tag{7}$$

$$R_{I3} = 8 \cdot B_P \cdot \frac{1s}{t_P} \tag{8}$$

To ensure that `Iperf3` is a suitable tool for our methodology, we conducted an experiment in which we measured how accurately the application is able to generate packets when a certain spacing between them is desired. For that investigation, we configured `Iperf3` to send frames of 1250 B with a target spacing of 100 μs. Using Equation (7) and (8), the aforementioned parameters were achieved by configuring a target bitrate of 94.4 Mbit/s and a packet size of 1180 B. To note, we also use the CBS qdisc configured accordingly. The results of this investigation are shown in Figure 9. In most cases, the interval between the sent packets does not deviate by more than ± 3 μs from the target 100 μs. The remaining outliers are not significant in the scope of our experiments, since we focus on the end-to-end latency of the system.

2) NETPERF

`Netperf`[6] is a utility for testing and verification of network performance. The tool supports numerous protocols, including Transmission Control Protocol (TCP) and UDP, and various modes of operation that enable uni- and bi-directional (round-trip, denoted as RR) testing. `Netperf` consists of two applications: a client and a server. The client connects to the server and is the one that accepts configuration for the type and details of the test to be performed. The server's role is to adhere to the client's requests and react appropriately. The tool provides numerous test-specific settings, enabling the configuration of amongst others:

- Number of test iterations
- Inter-packet/burst interval
- Packet/burst byte size
- Conversation TCP/UDP ports
- Central Processing Unit (CPU) affinity
- CPU performance measurement

`Netperf` provides numerous configuration options that could be beneficial for the experiment design. However, it has one major limitation preventing us from using it within the introduced methodology. The lowest setting inter-packet/burst interval is 1 ms which is too high for many of the traffic patterns introduced in Table 3 we aim to utilize.

[6]https://hewlettpackard.github.io/netperf/doc/netperf.html, Accessed 06.07.22

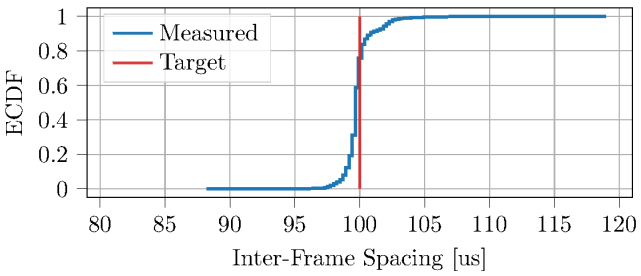


FIGURE 9. Measured Inter-Frame Spacing for `Iperf3` set for a target of 100 μs.

3) MoonGen

`MoonGen` [53] is an open-source software tool based on the high-speed packet processing framework DPDK.[7] It is designed to be used as an efficient packet generator. The utility can create and send packets at a rate of more than 10 Gbit/s using minimum-sized UDP packets. That bitrate corresponds to a packet rate of roughly 14.88 Mpps. The tool achieves this by generating packets via user-defined Lua scripts run utilizing the full potential of a single CPU core. To enable high-precision packet generation, MoonGen supports timestamping assisted by the hardware clock of the NIC. The integration with the hardware clock enables the tool to achieve sub-microsecond accuracy of packet generation. This accuracy is further enhanced by PTP. While `MoonGen` is a powerful tool, its integration within the methodology would require extensive implementation effort to support all required traffic patterns. Since other tools are available, we do not employ `MoonGen` within our experiments.

4) SEND UDP

`send_udp` is a custom application implemented in the C language. The tool is heavily influenced by the `udp_tai.c` first introduced by Tx Tools.[8] The application relies on the socket API and utilizes the `sendmsg` and `recvmsg` functions enabling custom packet generation and reception. Furthermore, it makes extensive use of several socket flags, most notably the `SOF_TIMESTAMPING_TX_HARDWARE` which enables hardware timestamping of generated and received packets within the application. Besides, `send_udp` further supports modification of packet metadata and payload data. Especially the ability to modify the payload content allows for entering the sequence number of a given packet and additional timestamps on the system taken before the packet is sent. A crucial functionality enabled by modification of metadata is the configuration of packet priority directly within the application. Therefore, we can use the application to generate traffic according to specific traffic patterns respecting the

[7]https://dpdk.org, Accessed 17.01.22

[8]https://gist.github.com/jeez/bd3afeff081ba64a695008dd8215866f, Accessed 05.07.22

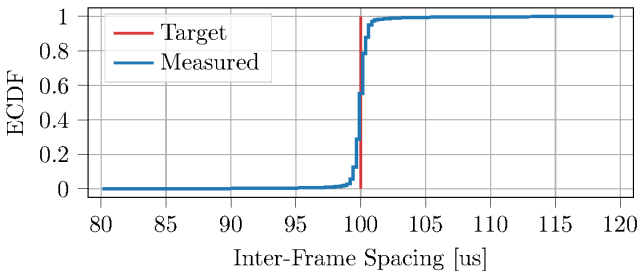


FIGURE 10. Measured Inter-Frame Spacing for send_udp set for a target of 100 μs.

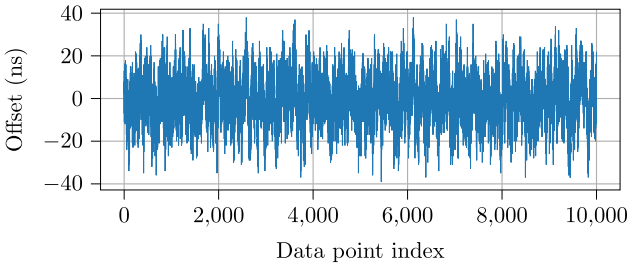


FIGURE 11. Intel® I210 NIC offset fluctuations.

given priority. In comparison to `Iperf3`, it has the ability to define a precise timestamp in the `cmesg` at which the NIC places the packet on the wire. This parameter enables the use of the Intel® LaunchTime feature present in Linux Kernel as ETF introduced in Section II-C1. Similarly, if we want to use TAPRIO qdisc without the TxTime mode, we need the `send_udp` application, which can specify when the packet should be dequeued with respect to its window. This might be of relevance when TAPRIO is used in the offload mode, as TxTime and offload mode are mutually exclusive [22].

Due to the custom packet creation process, `send_udp` is unable to generate packets at as high of a bitrate as, e.g., `Iperf3`, and therefore both applications are used extensively within the proposed methodology. Especially, for experiments with TAPRIO and ETF over multiple hops, we rely on the usage of TxTime mode. Otherwise, we would need to use a custom forwarder on each hop similar to `send_udp` to specify the packets dequeue time.

Figure 10 shows the inter-frame spacing for the commonly used scenario where we generate packets every 100 μs with payload size of 256 B, generating 20.48 Mbit/s. The packet inter-framce spacing does not deviate by more than ± 2 μs from the targeted 100 μs. Similarly like for `Iperf3`, the remaining outliers are not significant in the scope of our experiments, since we are focusing on the system performance in terms of the end-to-end latency.

B. SYSTEM PRECISION

In our experiments, we mostly use the Intel® I210 NIC. For the setup purely relying on software measurements we want to assess the precision it offers. To note, the I210 and other NICs offer hardware support that achieves higher precision due to hardware timestamping capabilities. The main challenge with precision measurements is to compare two values at the same node. To get an understanding of this error, we execute a specific experiment in which we synchronize a PHC to `CLOCK_REALTIME` and compare it to the exactly *same* PHC via `phc_ctl`. This experiment provides details on two possible origins of an error — the approximate comparison operation of `phc_ctl` and the synchronization of a PHC to another clock, in this case the `CLOCK_REALTIME`. Both of these errors are the main source of imprecision.

The `phc_ctl` tool is bundled with `linuxptp` [16]. It offers a comparison operation to query the system clock offset. The value is measured relative to a given PHC using `ioctl` system calls. With the Linux kernel 5.0 or higher, three methods offering varying accuracy are supported in the form of `ioctl` system calls to a PTP character device [54]. Our approach compares a PHC to `CLOCK_REALTIME` with `phc2sys` and `CLOCK_REALTIME` to a second PHC using `phc_ctl`. Also, the `phc2sys` uses the same `ioctl` system call as the `phc_ctl` for the system clock synchronization and is built into `linuxptp`.

Figure 11 shows the offset fluctuations of ± 40 ns for 12000 data points, where a data point is taken every 0.05 s. We deduce that the error is predictable in our system, but can potentially skew the results making a comparison on a a small scale more challenging. Yet, for the results we mostly look at values in the microsecond range, being the order of magnitude required by the SR class requirements.

C. SYSTEM OPTIMIZATION

The hardware and software of the network deployment need to fulfill the requirements outlined in Section II-A. To ensure the deterministic nature of experiments, we perform several experiments that verify the system's proper functionality. During that verification, we encountered several artifacts stemming from the COTS and open-source nature of the *EnGINE* framework deployment, requiring several optimizations of the system. In the following, we outline the encountered challenges and describe the necessary optimizations applied within the framework.

1) OPTIMIZATION METHODS AND TOOLS

To enhance the determinism of the SUT we utilize several tools and methods which are provided within the Linux ecosystem. In the following, we describe those approaches and outline the functionality they bring into the methodology.

CPU Affinity describes the ability of the Linux scheduler to enables manual assignment of execution threads to individual CPU cores [55]. The literature distinguishes between two affinity types: "hard" affinity and "soft" affinity. With hard affinity, a CPU core is defined that the thread is explicitly bound to. Such thread allocation is strictly respected by the CPU scheduler. The scheduler will never move the thread to a non-explicitly selected CPU core when hard affinity is used. With soft affinity, the CPU scheduler is less strict while keeping the threads running on the specified CPU cores. While the scheduler will try to keep the thread running on the same core

over its run-time, there is nothing stopping the scheduler from moving the process onto different CPU cores. This type of affinity is generally used within modern operating systems. In Linux, the affinity of a thread is configured using the `taskset`[9] tool. This utility allows to set into which CPU cores the process with its process id is assigned. To achieve that functionality, `taskset` employs a bitmask where each bit corresponds to a CPU core. It is furthermore possible to set affinity of hardware Interrupt Requests (IRQs). The principles of IRQ affinity are the same as for threads, the methods of the IRQ binding to a CPU core are different. The allocation is achieved by setting the `smp_affinity` mask of the IRQ to the desired cores.[10]

CPU Isolation describes the ability to isolate pre-defined CPU cores from the influence of the system scheduler. The isolated CPU cores are excluded from the Symmetrical Multiprocessing (SMP) balancer and the task scheduler is prevented from automatically placing any user or OS threads on them [56]. Despite not being able to place any tasks on the isolated cores, the processes can still be allocated manually to these processing units using CPU Affinity. Therefore, CPU Isolation enables a preparation of an enclave into which experiment-relevant tasks and hardware IRQs can be placed and protected from the influence of other functions which run on the system. In Linux, the CPU isolation is achieved using the `isolcpus`[11] system boot parameter.

Low-latency Ubuntu Kernel utilizes a flavor of the Linux kernel that is tailored for operation in environments where precise timing is required. This version of the kernel introduces several optimizations which enable the elimination of various influences induced by the SMP and system scheduler [57]. Main feature that supports the timing-optimized operation within the low-latency kernel is the consideration of hardware IRQs as threads. The threaded IRQs enable the configuration of their priority within the system scheduler. For our approach, this setting is essential for the operation of NICs. Threaded IRQs enable configuration of Real-Time (RT) priority for the IRQs of HW queues of the NICs. Another relevant innovation of the low-latency Ubuntu kernel are preemption points. This feature forces the task scheduler to intently search for threads with high priority and enable their execution before lower-priority tasks. The downside of preemption is that such execution of high priority functions may interrupt the run of low priority tasks. In Linux, the thread and IRQ priority is configured using the *chrt*[12] utility.

CPU Configuration and Power Management are powerful options enabling further optimization of system's performance. In the following, we introduce three options helping optimize the low-delay and low-jitter system operation.

Starting with the Simultaneous Multi-Threading (SMT) [58] which introduces execution of multiple (usually two) threads in parallel on a single hardware CPU core. The technology achieves this by keeping multiple architectural states in each core, while maintaining only a singe iteration of the computation hardware. From the OS perspective, this means that separate ''logical'' cores which can execute tasks at the same time are available, despite only one hardware core being present on the CPU. The expectation for SMT is the improvement of general CPU performance via a more efficient utilization of each of its cores.

Another CPU feature impacting its performance is the Automatic Overclocking (AO), in Intel® CPUs known as Turbo Boost [59]. This CPU option enables a dynamic adaptation of its clock frequency under high load scenarios with the goal of improving performance while increasing CPU's power consumption. While AO increases system performance and can potentially reduce task execution delay, it may have negative side effects such as increasing the system's jitter.

Final CPU option is the configuration of the Dynamic Voltage and Frequency Scaling (DVFS) governors. These CPU governors control the frequency at which the clock runs (excluding overclocking) and can be set to enable dynamic scaling of the frequency or to a pre-defined value. Within Intel® processors considered in this methodology, the DVFS governors are limited to two settings[13]: `powersave` and `performance`. The most significant difference between those two modes is that the `performance` mode will not consider the energy-saving features of the CPU. These features in `powersave` mode, may include lowering the CPU frequency when no load is present or putting the CPU into sleep states. The default configuration in Linux involves the CPU configured in `powersave` mode.

2) ELIMINATION OF OBSERVED SYSTEM ARTEFACTS

During verification of the fulfillment of SR classes requirements outlined in Table 1, we observed periodically occurring increases in the end-to-end delay, in the following referred to as delay spikes. This discovery prompted an investigation into the reasoning behind these occurrences. As a first step, to confirm the periodicity of the observed delay spikes, we conducted a 16-minute experiment EX_{SP1} using `Iperf3` as a traffic generator in a 3-hop scenario. The traffic was configured for a frame size of 1250 B and a frame spacing of 100 μs. The experiment was performed with the CBS configured on all interfaces along the path of the flow.

We then verified the end-to-end delay over the duration of the experiment. In this case, we calculate the delay as the time difference between when the packet is sent from the source and arrives at the sink. Such calculation is enabled

[9] https://man7.org/linux/man-pages/man1/taskset.1.html, Accessed 05.07.22

[10] https://www.kernel.org/doc/html/latest/core-api/irq/irq-affinity.html, Accessed 06.07.22

[11] https://www.linuxtopia.org/online_books/linux_kernel/kernel_configuration/re46.html, Accessed 06.07.22

[12] https://man7.org/linux/man-pages/man1/chrt.1.html, Accessed 06.07.22

[13] https://www.kernel.org/doc/html/v4.19/admin-guide/pm/intel_pstate.html, Accessed 06.07.22

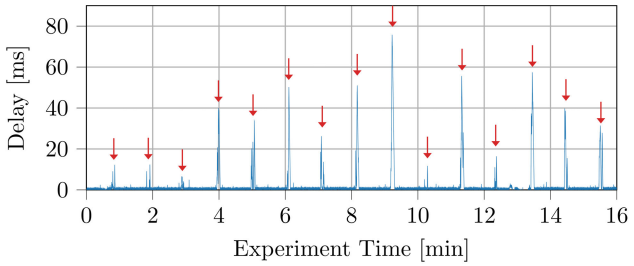


FIGURE 12. Delay over time for EX_{SP1} with observed periodic spikes.

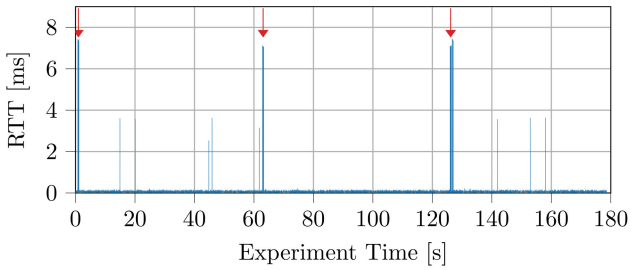


FIGURE 13. RTT over time for EX_{SP2} with observed periodic spikes.

due to the high timing precision of the system as described in Section V-B. The results can be observed in Figure 12. During the general experiment run, the delay stays below the 2 ms level required by SR class A. However, we observe periodic spikes in the delay occurring roughly every 60 s.

To ensure that the observed delay spikes are not caused by the behavior of the `Iperf3` traffic generator or the application of the CBS qdisc, we performed an additional experiment EX_{SP2}. The second experiment used the ping utility to measure the Round Trip Time (RTT) between two directly connected hosts. Ping was run with timestamp functionality enabled, requests being sent every 100 μs over the course of 180 s. The output of the utility was collected into a text file and subsequently evaluated and visualized.

The results of experiment EX_{SP2} are shown in Figure 13. We generally observe a very low RTT of under 0.5 ms, which is to be expected between directly connected hosts. However, again we observe the delay spikes occurring roughly every 60 s, confirming that some system behavior causes the spikes.

We considered a few hypotheses as to what system behavior could be causing the spikes. However, investigating the possible causes did not result in a clear answer. We looked at the following parameters and behavior of the system:

- Periodic system service – Verification of the syslog and CPU usage monitoring did not show in any correlation
- Network reconfiguration – No network reconfiguration found during the experiment
- PTP synchronization – `ptp4l` reconfigurations found, however, shown no clear correlation to delay spikes
- Power supply issues – No artifacts in system power use/supply found

While these investigations did not yield any results, we strongly suspected that the delay spikes were caused by a system task/service that is run periodically and may interrupt the execution of the applications or the operation of the NIC interrupts. To eliminate these influences, we applied CPU Isolation to isolate CPU cores from the system scheduler. Furthermore, we used CPU affinity to isolate system functions that are critical to the performance of the experiment from any other tasks ran within the OS. Both of those approaches are enabled by the use of a low-latency kernel. The following functions were isolated:

- Interrupts originating from the NICs
- Traffic generators such as `Iperf3` or `send_udp`
- Traffic sinks for `Iperf3`

We considered additionally isolating the packet capturing `tcpdump` application, however, the packet timestamps are recorded within the kernel and the CPU isolation for this tool would not benefit its performance.

To verify the functionality of our system optimization, we conducted the experiment EX_{SP3} using the same configuration as in EX_{SP1}. The experiment was run for 3 minutes. Results of EX_{SP3} are presented in Figure 14. The outcomes of the experiment show that the delay spikes were eliminated, with the end-to-end delay oscillating around 0.5 ms throughout the experiment run. We observe some fluctuation in the measured delay over time, however the jitter values in EX_{SP3} are under 125 μs in most cases what satisfies the requirements of the high priority SR class A.

While we were unable to identify the exact cause of the delay spikes, we argue that they were caused by some periodic system service that is run roughly every 60 s. We could eliminate the periodic delay increases with a low-latency Linux kernel and the usage of CPU Isolation and CPU affinity with critical functions needed for the experiment.

3) SYSTEM AND CPU PERFORMANCE OPTIMIZATION

To further optimize the experiment execution, we investigated a few combinations of the three CPU Configuration and Power Management options. The considered option combinations correlated with the experiment name are outlined in Table 6. The default system configuration is represented by experiment EX_{SO1}, where AO and SMT are enabled, but the DVFS governor is set to `powersave` mode. We only test a subset of all possible configurations, since some of the options, e.g., DVFS governor in `performance` mode, or the use of AO improves the obtained results. For all these experiments, the system optimization parameters resulting from Section V-C2 were applied. Similarly as in Section V-C2, the traffic was generated using `Iperf3` and configured for a frame size of 1250 B and a frame spacing of 100 μs. The packets traversed a 2-hop network with CBS adequately configured for 1250 B frames every 100 μs on all interfaces along the path of the flow. Each of the four experiments was run for 10 s.

The results of the individual CPU configuration option experiments are presented in Figure 15 and 16 with Figure 15 showing the measured delay over experiment time and Figure 16 presenting an ECDF of the measured jitter.

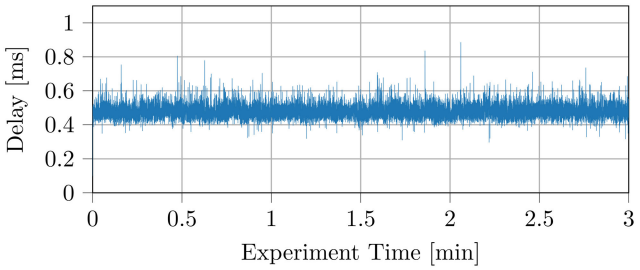


FIGURE 14. Delay over time for EX_{SP3} with system optimization applied.

TABLE 6. Combinations of investigated CPU configuration option combinations.

Experiment	AO	SMT	DVFS governor
EX_{SO1}	enabled	enabled	`powersave`
EX_{SO2}	disabled	disabled	`performance`
EX_{SO3}	enabled	disabled	`performance`
EX_{SO4}	enabled	enabled	`performance`

Figure 15a shows the observed delay for experiment EX_{SO1} using the default Linux configuration. We observe that the delay oscillates in the range of 0.15 ms to 0.35 ms (with some outliers) as denoted by the dashed lines. For the jitter shown in Figure 16a we observe values anywhere from -200 μs up to 240 μs with the majority of them within the range of -50 μs to 50 μs.

For experiments EX_{SO2}, EX_{SO3}, and EX_{SO4} with non-default configuration, we compare their results against the default Linux configuration applied in EX_{SO1}. In EX_{SO2} we observe that the disabled Overclocking and SMT negatively impact the upper bound of the delay as shown in Figure 15b with outliers of up to 0.7 ms. These settings also negatively impact jitter shown in Figure 16b with some of the observed values exceeding 300 μs.

With AO additionally enabled and still disabled SMT, we notice much better jitter in EX_{SO3}. While majority of the values remains in the -50 μs to 50 μs range, the lower bound is raised to -60 μs as indicated in Figure 16c. This observation implies that with disabled SMT we can achieve a better system performance in terms of jitter. However, the improvement does not apply to EX_{SO3}'s delay shown in Figure 15c. While the delay is more consistent due to the lower jitter, the overall latency is higher than in other experiments, with all values being greater than 0.2 ms.

With all CPU options enabled and DVFS governor set to `performance` mode, EX_{SO4} shows similar performance to EX_{SO1} concerning the delay shown in Figure 15d. While there are some outliers, the delay has a clearer lower bound at around 0.15 ms compared to that shown in Figure 15a. The jitter of EX_{SO4} presented in Figure 16d also performs similarly to the default Linux configuration, with its lower bound of -140 μs being better than that of EX_{SO1}.

With these observations, we conclude that the configuration introduced with EX_{SO4} improves the system performance compared to the default Linux configuration and performs best out of all investigated configuration combinations. Therefore, we perform all subsequent experiments with AO and SMT enabled, together with the DVFS governor configured for `performance` mode.

D. SETUP PARAMETERS

To achieve the required determinism and enable the TSN experiments, the parameters configured for each experiment must match and consider the system's capabilities. These parameters are configured system-wide and concern, for example, the NICs or time synchronization and include the associated supporting applications, functions, and hardware. This is especially relevant for ETF and TAPRIO as their settings are heavily system dependent and do not only rely on the expected traffic patterns.

1) QDiscs DEFAULT PARAMETERS

For a proper TSN operation, the qdiscs need to have an adequate configuration that considers the capabilities of the system and the expected traffic. **ETF**, as introduced in the Section II, has an adjustable parameter `delta`. To identify the optimal value of `delta`, we can measure the base latency of the system using the `cyclictest` [60] application. It measures the latency of a thread's intended and actual wake-up times. We run `cyclictest` on the ZGW and VCC hosts with the priority for SR class A for 60 s. The results show smaller maximum values when using the lowlatency kernel with optimizations from Section V-C2 applied. For both node types we observe an average of 3 μs and a maximum for individual threads below 100 μs. On the other hand, the standard kernel showed the maximum values of over 300 μs. To note, the average and maximum values depend on the system's load. Using higher values of `delta` lowers the risk of dropping a packet before it is dequeued.

The `delta` is a "fudge" factor accounting for various system delays. In case the selected value is too high, the packets dequeue operation is delayed, introducing additional latency. On the other hand, if the `delta` value is too low, delays are lower, but packets have a higher probability of being dropped. Of note, in case the TxTime is expired the packets are dropped immediately. Therefore, selecting a proper interval is important for the end-to-end delay, jitter, and packet drops. The performance is also affected by the current system utilization caused, e.g., by other running applications. Therefore, using the OS optimization techniques improves the stability significantly.

We investigated the impact of `delta` value in detail for our two available hardware setups representing the ZGWs and VCCs. We executed experiments with 25 μs steps and observed latency, absolute jitter values, and packet statistics. For all of the experiments 300 000 packets with inter-frame spacing of 100 μs and payload size of 256 B are sent. The packet statistics include the amount of captured packets (out of 300 000 packets), the number of packets lost between source and sink once placed on the wire, and the drop rate using the captured and sent packets data.

Figure 17 shows the overview of the results for 25-300 μs `delta` values for ZGWs setup, where ETF functionality is

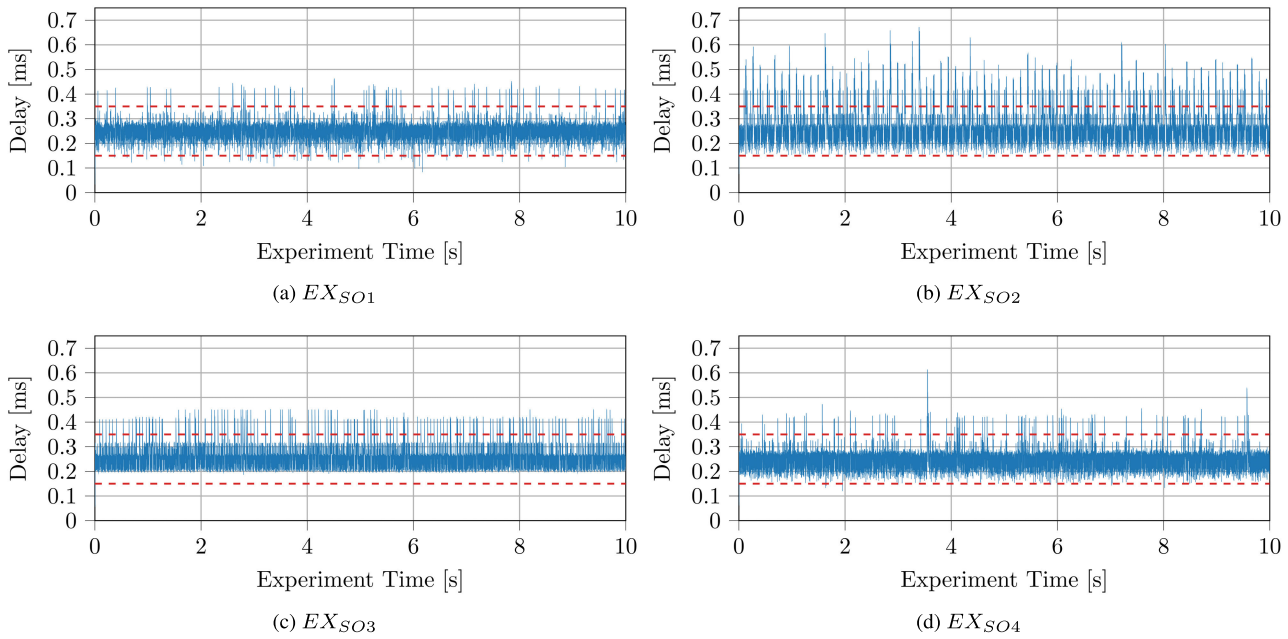


FIGURE 15. Delay over experiment time for CPU performance optimization experiments. Horizontal dashed lines represent the range where we expect most of the measured values based on EX_{SO1}.

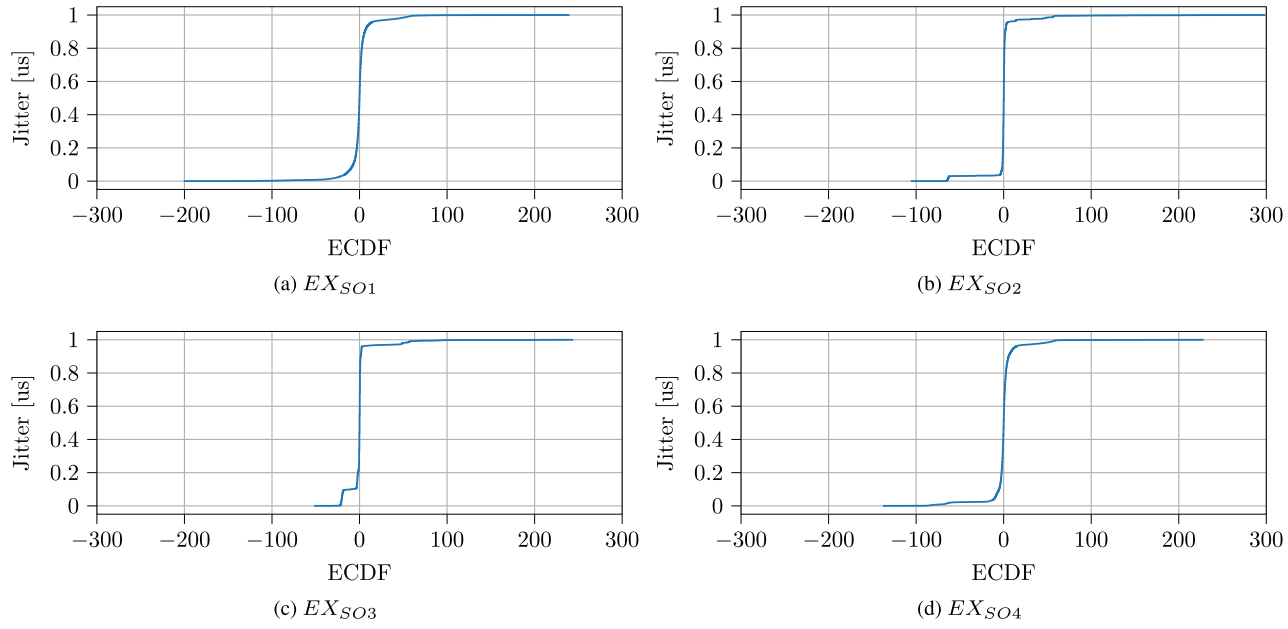


FIGURE 16. ECDF of Jitter for CPU performance optimization experiments.

either not offloaded or offloaded to the NIC. The delay and jitter figures show mean values along with 1st and 99th percentiles. For the case of no-offload, the results show similar behavior for delay and jitter with minimal fluctuations as shown in Figure 17a and Figure 17b respectively. The lowest values can be observed for 99th-percentile for `delta` value of 175 μs, where delay is 0.0226 ms and jitter 0.4768 μs. The 1st-percentile shows similar values as the mean. On the other hand, the offload shows a constant increase of delay with higher `delta` values as well as the jitter values. The jitter values fluctuate between -10 to 10 μs. Figure 17c shows drop rate, which behaves similarly to the experiments without offload. The number of captured packets for this value is 299 326 with a droprate of 0.0022. The dropped packets are always the first packets that are being sent. This behavior is verified by evaluating the sequence numbers stored in the payload. We assume those packet drops are caused by the ramp-up phase of the `send_udp` service, where the connection has to be prepared or the TxTime is missed.

As the results for ZGWs and VCCs are similar, we decided to showcase only those for ZGW. Since the `delta` value is also relevant for TAPRIO, we consider the values of 175 and

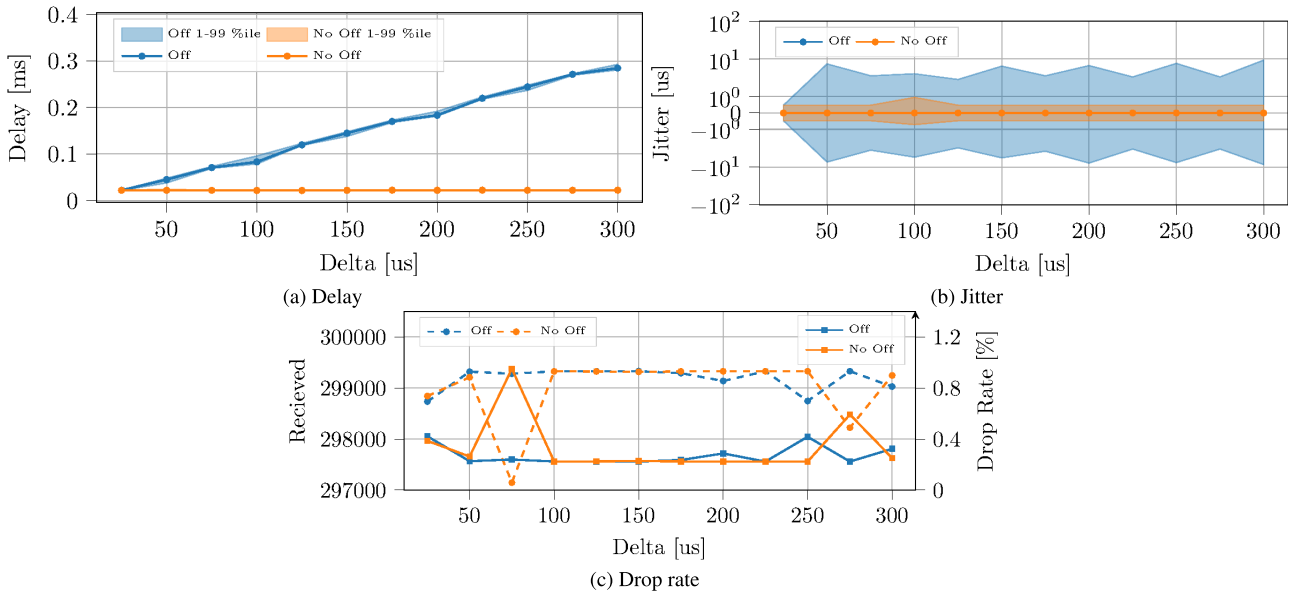


FIGURE 17. **ETF** `delta` **values with and without offload, delay and jitter mean with 1th and 99th-percentiles for various** `delta` **values, and packet received and droprates. Colored area represents the range between 1st and 99th percentiles values.**

275 μs that perform the best for the case with no offload. With offload, these values perform similarly with respect to jitter and delay, but we can see an increase in delay. Therefore, 175 μs is a preferred option. Next, we evaluate these values together with the TAPRIO parent qdisc to determine the best parameter for future experiments. The 275 μs is a more conservative value, but could account for fewer packet drops caused by too tight TxTime value.

TAPRIO itself classifies packets into traffic classes. We use TAPRIO in the TxTime assisted mode, i.e., TAPRIO configures a TxTime for each packet. This configuration is important when we want to use it together with `Iperf3`. A configuration parameter dependent on the performance is the `txtime-delay`, that accounts for the maximum delay between TAPRIO qdisc and the NIC. If a child ETF qdisc with activated offloading is installed for a traffic class, packets are sent precisely at the TxTime. Since this is the case in our experiments using TAPRIO, we name the TxTime mode as ETF assisted mode. For this configuration the child ETF qdisc must be configured with `skip_sock_check`. Otherwise the ETF would drop packets coming from TAPRIO.

For TAPRIO evaluation, we also need to configure the individual gates' opening times. We define a cycle time of 1 ms with three windows. First window is 300 μs wide and is used for the SR class A traffic with ETF child qdisc configured in the strict mode. Second window is 300 μs wide and is used for the SR class B traffic with ETF child qdisc configured in the deadline mode. The last window is 300 μs and is used for remaining traffic without any qdisc configured. For our experiments, we use a single `Iperf3` stream sending packets every 100 μs and payload size of 256B. The experiment runs for 10 s resulting in approximate 100 000 packets. This is due to the number of experiments and the configuration and processing time, as a longer evaluation period results in longer processing. Besides, we believe many packets should provide sufficiently strong statistics about the performance and trends. Since our target is to assess the performance over seven hops, we assess the `txtime-delay` and `delta` values over seven hops with and without HW offload.

Figure 18 shows the evaluation of the various `txtime-delay` over a single hop with the two pre-selected `delta` values with and without offload. Similarly, like for the ETF, the line represents the mean with 1st and 99th percentiles. Of note, the values of `txtime-delay` must be always greater than the `delta` value. We evaluate values in range of 200 to 300 μs with steps of 25 μs due to proximity of the delta values, then from 300 to 550 μs with steps of 50 μs, and 180 as a closer value to the `delta`.

Figure 18c shows the number of packets received and a drop rate. In this specific example, we see 0 packet loss on the wire with minor differences in the success rate. Even though we are sending 100 000 packets, we observe fewer packets is captured. These packet drops are most likely caused by a sudden load on the given node, but not in a continuous trend because of too strict parameter selection.

Figure 18b assesses the jitter, where the 99th percentile varies between 6.9141 to 11.2057 μs for the case of no-offload and 6.1989 to 14.782 μs for offload. The 1st percentile shows lower values than the 99th percentile, showing slight asymmetry. In contrast, the mean jitter values are low, but the 99th percentile shows higher values which might become more prominent over a larger number of hops. Finally, Figure 18a presents the delay values for those configurations, we see that all delay values are approximately 0.0422 ms for the no-offload scenario. On the other hand, with offload, we see an increase of values starting from

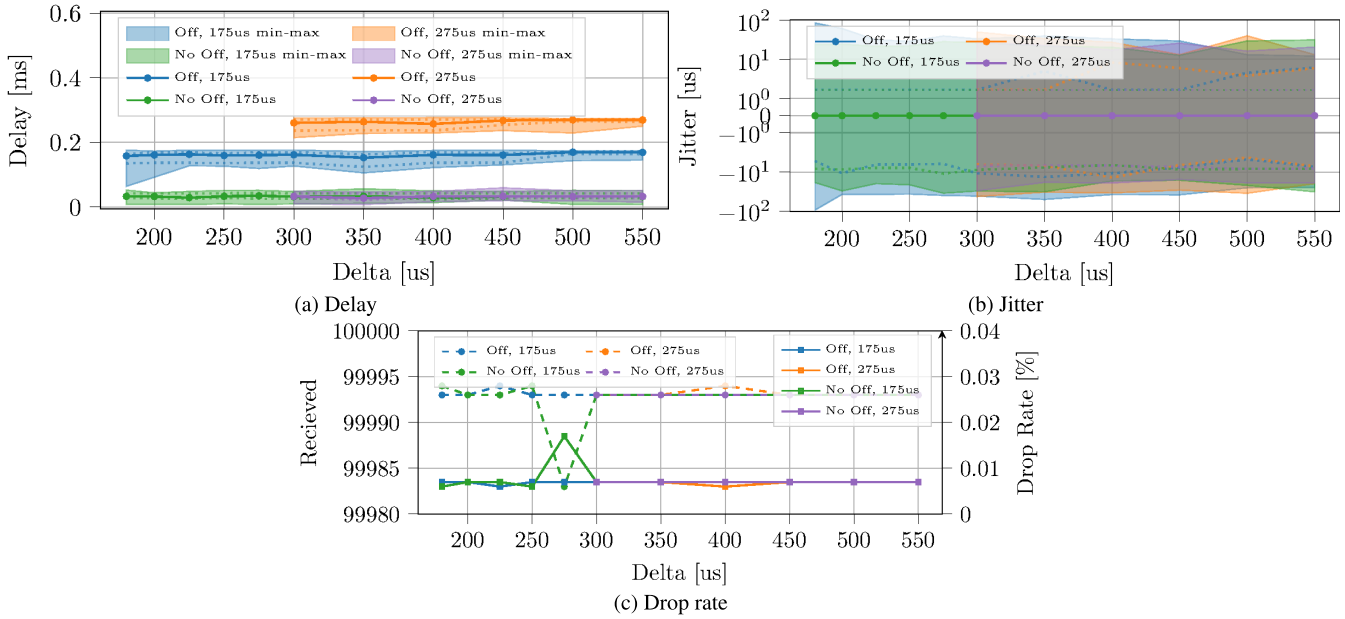


FIGURE 18. **TAPRIO** `txtime-delay` **and ETF** `delta` **values with and without offload, delay and jitter mean with 1st and 99th-percentiles for various** `delta` **values, and amount of received packets and drop rate. Colored area represents the range between 1st and 99th percentiles values. Off. - offload.**

0.1693 to 0.2725 ms. Especially interesting is to observe the behavior with different `delta` values for the same `txtime-delay` values, as the 100 μs is added to the overall delay. This confirms the findings regarding the offload and no-offload behavior of ETF. Therefore, we closely consider the `delta` value for the upcoming experiments. The 1st-percentile for jitter values is further from the mean than the 99th percentile.

To summarize the described approach, we choose the `delta` value of 175 μs and `txtime-delay` of 200 μs. The values for all of the experiments are similar, so choosing a lower value can especially positively impact the multi-hop scenarios. Regarding the ETF offload and no-offload mode, there is no significant difference in jitter, but delay plays in favor of no-offload. For future experiments we consider both offload and no-offload mode for the ETF with TAPRIO in TxTime assist mode to see its impact over a larger number of hops. When selecting higher `txtime-delay` we would observe a constant increase in delay, as it is automatically added to the given packet TxTime in case of the ETF in strict mode. Offload, as introduced for the ETF, increases delay but is better for the sake of stability. Worth of a note, that these parameters are system dependent, and also, if the window sizes are changed, the outcome might vary concerning the observed KPIs. Therefore, the results of the experiments are applicable to the current setup within our *EnGINE* framework but might vary even if the nodes are run under a higher CPU load. We dive into details of possible TAPRIO experiment traffic artefacts in Section VII.

For completeness, we also mention **CBS** as introduced in Section II-C3. The qdisc requires *idleSlope*, *sendSlope*, *hiCredit*, and *loCredit* parameters for its configuration. Since these settings are mostly system independent, except available NIC bitrate, and only rely on the expected traffic patterns, Equation (2) to (5) can be used to calculate the four parameters without experimental investigation. Of note, the priorities of the configured CBS qdiscs matter, as the lower priority configuration needs to consider the interfering traffic of the higher priority one.

2) PTP CONFIGURATION

Within our methodology, we use the `linuxptp` project for PTP. As we are mainly using the Intel® I210 NIC, we need to consider the many PHCs as available ports. Therefore, we have to run the `ptp4l` in the Just a Bunch of Devices (JBOD) mode (`boundary_clock_jbod`), which ensures that individual PHCs get later synchronized on a single device. For the synchronization of the clocks, we use the `phc2sys` service that takes the information taken from the `ptp4l` and applies the configuration accordingly.

Listing 1 shows a configuration based on `gPTP.cfg` profile, with corresponding settings for each used interface. The PTP for the message exchange only utilizes Layer 2 and operates in the P2P delay mechanism. Besides, the configuration uses the two-step mode for synchronization. All nodes operate as the BCs or OCs. This can result in larger clock deviations over more hops [61], but we believe the clock performance does not significantly impact our measurements. The messages between the individual nodes are exchanged every $2^{-3} = 0.125$s based on the `logSyncInterval`.

Of note, challenges with the coexistence of PTP and TAPRIO require us to use a PTP in the form of an overlay network, where we have additional connections that allow for the clock synchronization.

```
[global]
gmCapable 1
priority1 248
priority2 248
logAnnounceInterval 0
logSyncInterval -3
syncReceiptTimeout 3
neighborPropDelayThresh 800
min_neighbor_prop_delay -20000000
assume_two_step 1
path_trace_enabled 1
follow_up_info 1
transportSpecific 0x1
ptp_dst_mac 01:80:C2:00:00:0E
network_transport L2
delay_mechanism P2P

[enp9s0]
boundary_clock_jbod 1
[enp10s0]
boundary_clock_jbod 1
```

Listing 1. Sample of gPTP.cfg.

E. NETWORK ARCHITECTURE

Figure 7 shows the underlying infrastructure provides a high flexibility with respect to connections and topologies. Within the scope of IVNs mesh like topologies are often discussed due to reliability aspects, where in case of a node or cable failure, the traffic can use a redundant path. For our experiment design and later on evaluation we mainly rely on the two types of topologies - the line and mesh topology. The line topology is relevant for the evaluation of the performance over various number of hops, which are relevant to fulfill the SR class requirements. On the other hand, the mesh topologies are assessed with the exemplary use-case that focuses on the cross traffic operation emulating the realistic topologies inside of vehicles. Worthy of a note, we focus on the Layer 2 evaluations of the individual TSN standards.

VI. EXPERIMENT DESIGN

Emulating a complex system that represents a real-world deployment of a TSN and IVN network requires a robust experiment design and preparation methodology. In this work, we distinguish between three types of experiments:

1) Environment validation
2) Configuration validation
3) Solution evaluation

The goal of the initial environment validation experiments is to show that the hardware deployment can fulfill all requirements and therefore support the investigated use-case. The subsequent configuration validation experiments check whether the network and qdisc configurations are correct and well understood. These further verify whether the system can support the requirements placed on it. Finally, the solution evaluation experiments investigate actual IVN or TSN use-case that can be proposed by the users of this methodology.

In the following, we describe how each of the three experiment types could look. These descriptions are prepared with the *EnGINE* hardware deployment in mind. However, we generalize them so that the proposed settings can be applied to any experimental environment that fulfills the same requirements as those placed on the *EnGINE* framework. Of note, the configurations proposed for TAPRIO and ETF require additional considerations when deployed on different hardware than that of *EnGINE* as outlined in Section V-D.

A. EXPERIMENTAL ENVIRONMENT CAPABILITIES

To ensure that the experimental environment can support the envisioned experiments, we prepare several scenarios verifying the capabilities of the physical deployment. Since in the scope of this work, we focus on IVNs and TSNs, these experiments are tailored towards verification of the fulfillment of SR class A and SR class B requirements outlined in Table 1. In the following, we introduce these experiments and outline the configurations used to execute them. Of note, we define a network hop as a link traversed between two network nodes. As an example, a 2-hop line network topology would include three nodes ZGW 6, ZGW 1, and ZGW 2 in that order and utilize the links numbered 1 and 2 between them as shown in Figure 19.

For the following experimental scenarios, we define two types of flows that can be placed within the network: *limited* and *unlimited*. For each of the two types, the traffic is generated using the `Iperf3` application. A *limited* flow has a configuration that enables it to send 1250B PHY size frames every 100 μs. Such configuration amounts to as bitrate of 100μs with `Iperf3` configured according to Equation (7) and (8). With these equations, the resulting relevant settings of the `Iperf3` client are:

- Bitrate $R_{I3} = 94.4$ Mbit/s
- Payload Size $B_P = 1180$ B
- UDP as the Transport Layer protocol

In contrast, the *unlimited* flow do not have any configuration limiting their sending bitrate. Therefore, in this case, the `Iperf3` client sends as many UDP maximum size packets as possible. In general, the client for each flow is always configured on the source node, while the server is present at the sink node. The packets are transmitted from the client to the server. In the following experiments, we utilize the network topology introduced in Figure 19, mapped to the physical *EnGINE* deployment introduced in Figure 7. We vary the number of traversed hops in the network from one up to seven, with ZGW 6 always considered as a source node. The sink for the flows is placed on the appropriate ZGW for the number of hops indicated by the number next to the link. As an example, with 4 hops network, the sink node is ZGW 5. For each experiment run we collect at least 100000 packets.

1) MQPRIO VALIDATION

We first verify the capabilities of the deployment in a non-overloaded setting with four flows placed along the same path in the network. The placement of the verification flow with the varying number of hops is shown in Figure 19. The

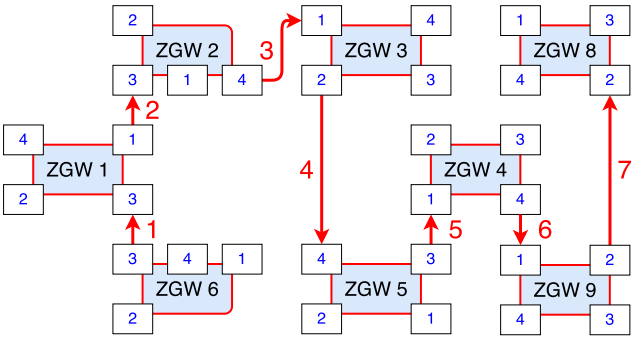


FIGURE 19. 1 to 7 hop experimental environment validation flow, as an example, placed within the *EnGINE* deployment.

primary goal of this EX_{MQ} scenario is to verify whether each of the queues presents within the NICs behaves in a similar way. Hence, we do not employ any TSN traffic shaping and test solely the functionality of the MQPRIO qdisc configured on each interface in the network. Furthermore, we run seven experiments in a campaign to verify whether the experimental environment can fulfill the requirements of SR classes over seven hops without any TSN shaping. With those experiments, we cover a range where the number of hops is varied from 1 up to 7.

For MQPRIO validation we run two experiment campaigns which differ only in the configuration of the flows generated by `Iperf3` clients:

1) EX_{MQ1} - All four flows are *limited* flows
2) EX_{MQ2} - Two flows (Flow 3 and 4) are *limited* flows placed on NIC queues 1 and 2. Two flows (Flow 1 and 2) are *unlimited* flows placed on NIC queues 3 and 4.

Experiment EX_{MQ1} aims to verify whether SR class A requirements are attainable without TSN qdiscs when each flow does not try to take all bandwidth for itself. EX_{MQ2} additionally aims to showcase why TSN traffic shaping is necessary to fulfill SR class A requirements when some flows do not respect bandwidth allocations.

2) CBS SINGLE FLOW VALIDATION

To verify the experimental environment's capability of supporting traffic of SR class A when CBS is used, the experiments within the second EX_{CS} validation scenario includes one flow traversing the network. Since the SR class's requirements need to be fulfilled over seven hops, the experimental campaigns in this validation scenario again utilizes the network shown in Figure 19 and consist of 7 individual experiments. In each case, we vary the number of hops covering all possibilities from one to seven.

As these EX_{CS} experiments are used to validate the hardware deployment, we are not following any previously introduced traffic patterns. Instead, we are using one *limited* `Iperf3` flow configured for PCP 3, corresponding to SR class A. The CBS is configured accordingly for the desired 1250B PHY size frames and the bitrate of 100Mbit/s. Using Equation (2) to (5), we obtain the appropriate values needed to set up the qdisc:

- $idleSlope_A = 100000$
- $sendSlope_A = -900000$
- $hiCredit_A = 155$
- $loCredit_A = -1125$

In EX_{CS}, the CBS qdisc for the flow is configured with the help of MQPRIO for VLAN Tag PCP 3 and associated with queue 1 of the NIC. Of note, the qdisc was not configured in offload mode. We observed that with hardware offload, the behavior of the NIC is non-deterministic.

We perform two experiment campaigns for the second EX_{CS} verification scenario. Experiment EX_{CS1} has CBS configured only on the source node, with all remaining nodes along the flow having no shaping applied. Experiment EX_{CS2} has CBS applied on all nodes along the flow. The goal of these experiments is to assess the impact the CBS qdisc has on the delay and jitter in various circumstances.

3) CBS MULTIPLE FLOW VERIFICATION

To verify the coexistence of multiple SR classes when the CBS qdisc is used, we further propose scenario EX_{CM} that combines certain aspects of the previous two, introduced in Sections VI-A1 and VI-A2. EX_{CM} utilizes the same network topology shown in Figure 19 as in previous experiments. The scenario includes the configuration of flows introduced in experiment campaign EX_{MQ2}, where two flows are *limited* and two are *unlimited*. The limited flows are placed on PCPs 3 (Flow 4) and 2 (Flow 3) corresponding to SR classes A and B respectively. Flows 1 and 2 are *unlimited* and are assigned to PCPs 0 and 1 respectively.

The CBS configuration for SR class A is the same as shown in Section VI-A2. For SR class B we again apply the Equation (2)to (5). Of note, the settings for SR class B do consider the traffic of SR class A as well. The resulting configuration is shown in the following:

- $idleSlope_B = 100000$
- $sendSlope_B = -900000$
- $hiCredit_B = 297$
- $loCredit_B = -1125$

As in Section VI-A2, we again distinguish between two experimental campaigns with EX_{CM1} having CBS configured only on the source node and EX_{CM2} having the qdisc on all interfaces along the flow. On interfaces where CBS is not present, we still configure MQPRIO witch each flow having its own NIC queue assigned.

4) TAPRIO VALIDATION

Within the Section V-D we provided a detailed steps how to find a suitable setup parameters for the ETF and TAPRIO. Mainly, the focus is on the ETF in the strict mode. In the upcoming experiments, we aim to assess the strict mode and the deadline mode, and BE traffic. Each of the different modes operates in its own window, which should result in minimal influence on each other. A target of the experiments

is to verify if the time division of various traffic priorities satisfies the defined requirements. In addition, we want to assess how the corresponding queues affect the performance and whether the TAPRIO configuration for window sizes achieves the targeted metrics. The experiments campaigns we present can be categorized as follows:

1) EX_{TS} - shows performance of ETF in deadline and strict mode with a single traffic flow over one to seven hops for SR class A and B
2) EX_{TM} - shows performance of ETF in deadline and strict mode with a three traffic flows corresponding to SR class A, B, and BE over one to seven hops
3) EX_{TS-T} - shows relation between traffic pattern and a window size, uses ETF in strict mode with a single traffic flow (SR class A) over seven hops
4) EX_{TS-W} - shows performance of smaller window sizes and cycle time with ETF in strict mode with a single traffic flow (SR class A) over seven hops

For better comparability among the experiments, we start with the same traffic patterns as in CBS experiments for Flows 4 and 3, which correspond to SR classes A and B, respectively. In comparison, Flows 1 and 2 that use the *unlimited* traffic are assigned to PCPs 0 and 1 respectively and share the same window allocated for the BE.

The experiments we plan to conduct are depicted in Figure 20 along with a sample configuration of the window cycle and the SRs classes mappings. EX_{TS} assesses and serves as the baseline with a single flow at a time either for SR class A or B. Besides, it evaluates the impact of the `txtime-delay` on the performance over seven hops compared to the single hop setup as done in Section V-D. EX_{TM} builds on top of the EX_{TS} and uses additional traffic categorized as policed and unpoliced traffic as already outlined. In addition, it provides insights into the impact of various window sizes, their shift over the hops, and the behavior of the offload functionality provided by the Intel® I210 NIC.

EX_{TS-T} analyses the outcomes of EX_{TM} and provides experiment results, where we assess the traffic pattern generation with a given window size with and without offload. Since all of the previous experiments relied on the traffic pattern generating packets each 100 μs with a window cycle of 1 ms, we show the behavior if we align the packet generation time to the window cycle time.

Finally, EX_{TS-W} shows if lowering the window cycle to 100μs with a shift on each hop as done in EX_{TM} is possible using the COTS hardware and open-source solutions.

B. EXEMPLARY USE-CASE

With a properly tested and validated experimental environment, solutions and use-cases can be evaluated. To better visualize such experiments, we propose an exemplary use-case. The experiment showcases a sample network topology, initially introduced in Figure 7 as a high-end vehicle network topology. We build upon this network, introducing five flow paths as visualized in Figure 21. Along each of the first

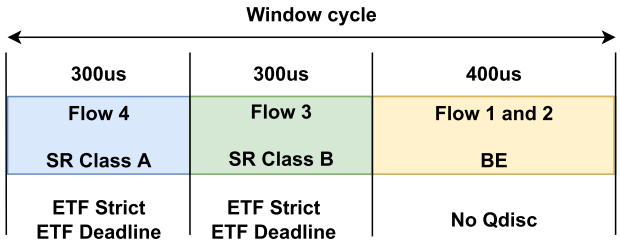


FIGURE 20. Example of TAPRIO window configuration and planned child qdiscs configuration with respect to SR classes.

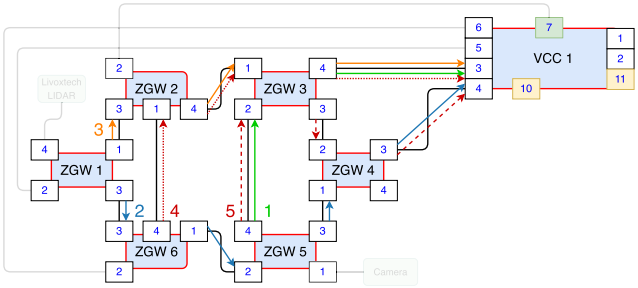


FIGURE 21. Network topology including flow paths for the exemplary use-case. Based on the high-end vehicle network topology from Figure 7.

three paths, we generally place traffic of one SR class and additionally include one flow with BE traffic. Flow path 1 follows from ZGW 5, through ZGW 3 and terminates at VCC 1 forming a 2 hop connection. Flow path 2 originates at ZGW 1 and continues towards VCC 1 via ZGW 6, ZGW 5, and ZGW 4, thus representing a 4 hop flow configuration. Flow path 3 also starts at ZGW 1 and flows through ZGW 2 and ZGW 3, terminating at VCC 1, being a 3 hop connection. The paths 4 and 5 only include some BE cross-traffic. Flow path 4 has its source at ZGW 6 and continues through ZGW 2 and ZGW 3 with its sink again on VCC 1. The last path starts at ZGW 5 and passes through ZGW 3 and ZGW 4 before terminating at VCC 1.

Along each of the flow paths, we configure a few individual flows as shown in Table 7. The traffic patterns are inspired by Table 3. Apart from the BE traffic, along flow path 1 we place C&C as well as ADAS flows on priority 3 corresponding to SR class A. Flow path 2 receives some LIDAR and RADAR flows, also associated with priority 3 and SR class A. The third flow path has three flow types: RADAR, ultra-sound (US), and GPS. These flows are associated with priority 2 corresponding to SR class B. The final two flow paths have only one BE flow each placed on priority 0. Such configuration aims to test the impact of cross-traffic on the delay and jitter of SR classes A and B.

The traffic was generated using the `Iperf3` application using the values outlined in Pkt. Size and Period columns of Table 7. These packet sizes and periods were then processed using Equations (7) and (8) to obtain the necessary settings used to configure the traffic generator.

This is an exemplary configuration using findings presented in the previous sections. Furthermore, we propose the used network topology based on the *EnGINE* deployment we

TABLE 7. Traffic Matrix for the exemplary use-case.

Flow Identifier	Source	Sink	Priority	Port	Pkt. Size [B]	Period [µs]	Bitrate [Mbit/s]	# Flows	Total Bitrate [Mbit/s]
				Flow Path 1					
$\mathbf{F1_{C\&C}}$	ZGW 5	VCC 1	SR-A (3)	2101	1024	1000	8.192	6	49.152
$\mathbf{F1_{ADAS}}$	ZGW 5	VCC 1	SR-A (3)	2102	1500	600	20	2	40
$\mathbf{F1_{BE}}$	ZGW 5	VCC 1	BE (1)	2103	1250	100	100	1	100
				Flow Path 2					
$\mathbf{F2_{LIDAR}}$	ZGW 1	VCC 1	SR-A (3)	2201	1500	2500	4.8	3	14.4
$\mathbf{F2_{RADAR}}$	ZGW 1	VCC 1	SR-A (3)	2202	1250	2500	4	3	12
$\mathbf{F2_{BE}}$	ZGW 1	VCC 1	BE (1)	2203	1250	100	100	1	100
				Flow Path 3					
$\mathbf{F3_{RADAR}}$	ZGW 1	VCC 1	SR-B (2)	2301	1250	2500	4	3	12
$\mathbf{F3_{US}}$	ZGW 1	VCC 1	SR-B (2)	2302	512	20000	0.2048	3	0.6144
$\mathbf{F3_{GPS}}$	ZGW 1	VCC 1	SR-B (2)	2303	256	10000	0.2048	1	0.2048
$\mathbf{F3_{BE}}$	ZGW 1	VCC 1	BE (1)	2304	1250	100	100	1	100
				Flow Path 4					
$\mathbf{F4_{BE}}$	ZGW 6	VCC 1	BE (0)	2401	1250	100	100	1	100
				Flow Path 5					
$\mathbf{F5_{BE}}$	ZGW 5	VCC 1	BE (0)	2501	1250	100	100	1	100
							Total:	**26**	**628.3712**

had access to. However, the described configuration can be adapted to any other deployment, assuming enough nodes and connections are available. This use-case exemplifies how a TSN and IVN experiment methodology works and can be used to obtain results for any solution one wants to evaluate.

In the following, we introduce two variations of the introduced use-case experiment. The first includes the CBS qdisc used to police the traffic of SR classes A and B. The second experiment uses TAPRIO and ETF to achieve the same effect.

1) CBS VARIANT

To prepare the qdisc configuration of the CBS variant of the use-case experiment, we again utilize the Equations (2) to (5). Due to the similarity of traffic flowing across multiple network interfaces, we only require four distinct CBS qdisc configurations to attain the necessary traffic policing for SR classes A and B. The configurations and their assignments are described in Table 8. The remaining priorities and queues were configured for BE traffic. Of note, we apply the configurations on a per-interface basis. On interfaces in Figure 21 not specified in the Table 8 we apply the MQPRIO configuration outlined in Section VI-A1.

We propose four experiments for the CBS variant configuration. In EX_{UCC} we use all five flow paths and their corresponding traffic as described in Table 7. We then perform an experiment set EX_{UCC-F}, consisting of three individual experiments EX_{UCC-F1}, EX_{UCC-F2}, and EX_{UCC-F3}. In each of the three sub-experiments, we only use one flow path with just the CBS policed traffic flows present in the network. In more detail, considering flows presented in Table 7, EX_{UCC-F1} includes flows $F1_{C\&C}$ and $F1_{ADAS}$. EX_{UCC-F2} uses flows $F2_{LIDAR}$ and $F2_{RADAR}$, while EX_{UCC-F3} utilizes flows $F3_{RADAR}$, $F3_{US}$, and $F3_{GPS}$.

2) TAPRIO AND ETF VARIANT

Similar to the CBS variant, we evaluate policed traffic for windows 1 and 2 for SR classes A and B respectively, based on the Figure 20. The last, third, window is allocated for the BE traffic. The configuration is applied per interface, according to the present flow paths. We apply the MQPRIO configuration outlined in Section VI-A1 on the interfaces not specified in the Table 8.

The EX_{UCT} experiment campaign assess the usage of strict and deadline modes of ETF qdisc. We use the same `delta` and `txtime-delay` values as specified in Subsection V-D and the results provided by EX_{TS}. To assess the impact of cross traffic on the hop between ZGW 1 and VCC 1, we run two experiment sets. EX_{UCT-F} does not include the cross traffic, which should serve as a baseline measurement for the given number of hops and traffic flow. Second, the EX_{UCT} setup uses the additional flows between ZGW 5 - ZGW 3 - VCC 1, with a corresponding policed traffic as outlined in Table 7. We can compare both results, but we focus on the cross traffic overhead, which is relevant for Flows 1 and 3 on the last hop towards the VCC 1.

C. EVALUATION METHODS

We mostly consider two evaluation metrics for the experiments introduced above: delay and jitter. These metrics are relevant for SR class requirement fulfillment. Due to the time-sensitive nature of the experiments, and the high system precision outlined in Section V-B, we can directly measure the delay between the source and sink nodes.

The measurements are performed by collecting two Packet Captures (PCAPs), one on the source and one on the sink of a network flow. The captures are done using the hardware timestamping feature of the NIC where possible. Since we

TABLE 8. CBS qdisc configuration for the CBS variant of the use-case experiment. Combination of node and interface in format ZGW X If Y for node X and interface Y corresponding to notation in Figure 21.

Type	Queue	PCP	idleSlope	sendSlope	hiCredit	loCredit
Configuration 1 *Applied on:* ZGW 1 If 1 and ZGW 2 If 4						
SR-B	2	2	12820	-987180	20	-1234
Configuration 2 *Applied on:* ZGW 1 If 3, ZGW 4 If 3, ZGW 5 If 3, and ZGW 6 If 1						
SR-A	1	3	26400	-973600	41	-1461
Configuration 3 *Applied on:* ZGW 3 If 4						
SR-A	1	3	89152	-910848	138	-1367
SR-B	2	2	12820	-987180	41	-1234
Configuration 4 *Applied on:* ZGW 5 If 4						
SR-A	1	3	89152	-910848	138	-1367

know that the source's and sink's clocks are accurately synchronized using PTP, we can then correlate the two PCAPs together. For correct correlation of the packets, we utilize the timestamp and sequence numbers included in the payload of packets generated by `Iperf3` and `send_udp` applications.

1) DELAY CALCULATION

After the packets from the source and sink are correlated, we calculate the end-to-end delay d_{e2e} as the difference between the time at which the packet was received at the sink t_R and sent on the source t_S. In the later sections, we abbreviate d_{e2e} as delay or latency.

$$d_{e2e}(X) = t_R(X) - t_S(X) - t_o \tag{9}$$

The exact calculation for packet X is introduced in Equation 9. Due to the way how packet timestamps are recorded in Linux, in certain cases there might be an additional time offset t_o that needs to be considered. Amongst others, this occurs when a system and NIC hardware timestamps are compared. In general the system time is given as Coordinated Universal Time (UTC) and NIC hardware time as International Atomic Time (TAI). The difference t_o for those two-time formats at the time of writing amounts to 37 s.

2) JITTER CALCULATION

The calculation for end-to-end delay jitter, that is the difference in delay between two subsequent packets, is made based on d_{e2e} introduced in Section VI-C1.

$$j_{e2e}(X) = d_{e2e}(X) - d_{e2e}(X-1) \tag{10}$$

Equation 10 introduces the calculation for jitter observed for packet X. It is defined as a difference of packet X's delay $d_{e2e}(X)$ and the delay of the previous packet $d_{e2e}(X-1)$. The jitter for the first evaluated packet $j_{e2e}(0)$ is ignored, as there were no packets before to consider.

In the following, we look at two jitter value representations. The first one is measured jitter which directly corresponds to the value obtained for $j_{e2e}(X)$. In the later sections we often refer to measured jitter as jitter. The second one is absolute jitter which takes an absolute value of the result of Equation 10. We differentiate between the two to provide more more detail with measured jitter where required and provide enough abstraction with absolute jitter to concisely introduce numerous results.

VII. RESULT EVALUATION

Based on the definitions introduced in Section VI, we prepared and ran these experiments using the *EnGINE* deployment. In the following, we showcase the obtained results and signify which metrics are relevant. In more detail, we show:

1) Delay and absolute jitter obtained during the validation of the MQPRIO qdisc
2) Delay and measured jitter recorded during the validation of the CBS qdisc in a single and multi flow scenarios
3) Delay and absolute jitter observed in the TAPRIO validation experiments
4) Delay and measured jitter when modifying the TAPRIO window placements, sizes, and traffic pattern
5) Delay and measured jitter from the experimentation involving the exemplary use-case with CBS qdisc applied
6) Delay and absolute jitter from the experimentation involving the exemplary use-case with TAPRIO qdisc applied

We furthermore show which KPIs provide valuable insights and outline how the results can be interpreted to assess system performance. Of note, the results below show a subset of attainable values and are specific to the *EnGINE* deployment. However, the presented methods and evaluations can be generalized to any system with comparable capabilities.

A. EXPERIMENTAL ENVIRONMENT CAPABILITIES

Using the methods introduced in Section VI-C, we start an evaluation of the capabilities of the experimental environment. We focus mainly on the fulfillment of SR class A requirements outlined in Table 1. The evaluation is based on the scenarios introduced in Section VI-A.

1) MQPRIO VALIDATION

We first evaluate the MQPRIO verification experiments EX_{MQ} investigating the functionality of the qdisc and its applicability for TSN experiments. The results of the first MQPRIO validation experiment EX_{MQ1} with all four flows generated with a bitrate of 100 Mbit/s. are shown in Table 9. We look at the mean, median, and 99th-percentile values of delay and jitter in these experiments for each of the flows individually. Of note, for jitter, we show its absolute value.

In this scenario, we use the values to assess whether the flows are treated equally across all queues. While there are some differences in delay between the flows and subse-

TABLE 9. Experiment EX_{MQ1} mean ($\bar{x}$), median ($\tilde{x}$), and 99th-percentile (99%ile) delay and absolute value of jitter for each flow.

Hops	Flow 1			Flow 2			Flow 3			Flow 4		
	$\bar{x}$	$\tilde{x}$	99%ile	$\bar{x}$	$\tilde{x}$	99%ile	$\bar{x}$	$\tilde{x}$	99%ile	$\bar{x}$	$\tilde{x}$	99%ile
Delay												
1	0.03	0.03	0.06	0.03	0.02	0.06	0.03	0.02	0.06	0.03	0.02	0.05
2	0.19	0.19	0.25	0.20	0.21	0.24	0.19	0.18	0.22	0.20	0.20	0.25
3	0.45	0.46	0.51	0.37	0.37	0.50	0.40	0.42	0.49	0.34	0.36	0.45
4	0.61	0.62	0.76	0.50	0.52	0.64	0.54	0.54	0.67	0.53	0.54	0.67
5	0.67	0.69	0.85	0.81	0.81	0.91	0.65	0.67	0.83	0.79	0.79	0.90
6	0.81	0.88	1.04	0.96	0.97	1.07	0.99	0.99	1.09	1.01	1.02	1.10
7	0.97	1.02	1.24	1.08	1.08	1.23	1.03	1.04	1.16	1.04	1.03	1.20
Jitter												
1	5.90	4.05	32.66	5.40	4.05	32.90	5.37	4.05	33.14	5.72	4.29	23.84
2	10.07	4.29	72.72	32.61	4.29	164.99	5.59	4.29	34.09	14.60	4.29	106.10
3	11.19	4.29	130.89	27.84	4.05	262.98	20.86	4.29	238.66	39.67	4.29	262.26
4	19.53	4.29	233.17	44.21	4.29	290.87	24.06	4.29	196.93	38.75	4.29	293.73
5	52.34	4.29	328.06	14.98	4.05	149.73	56.91	4.53	366.45	17.18	4.29	160.22
6	110.87	30.28	649.93	20.73	4.29	174.28	22.51	4.29	174.05	19.47	4.29	156.88
7	94.07	14.07	572.92	23.05	4.29	222.21	21.63	4.29	187.16	22.45	4.29	202.18

quently NIC queues, no clear pattern shows when the number of hops increases. We observe mean delay values as low as 0.03 ms for one hop experiments, with the highest observed average of 1.08 ms for Flow 2 in 7 hop experiment. The median delay closely follows the mean with the highest difference of 0.07 ms recorded for Flow 1 in a six-hop scenario. In most cases, the median is $\geq$ mean delay. In the 7 hop experiment, we record the highest 99%ile delay of 1.24 ms for Flow 1, meaning that MQPRIO alone with *limited* `Iperf3` configuration can satisfy the requirements of SR class A across all four flows.

Similarly, we do not observe many variations in the absolute jitter across all four flows recorded during experiment EX_{MQ1} shown in the bottom part of Table 9. The highest mean absolute jitter of 110.87 μs was observed for Flow 1 in the six hop scenario. Such values indicate that the experimental environment is capable of fulfilling the SR class A jitter requirement of 125 μs with just MQPRIO applied on all hops. However, we observed higher 99%ile absolute jitter values of up to 649.93 μs for Flow 1 in 6 hop scenario. Of note, all flows exceeded the 99%ile absolute jitter of 125 μs required for the fulfillment of SR class A requirements when more than two hops were used in the network.

The outcome of EX_{MQ2} presented in Table 10 is more varied. For this experiment, we again look at the mean, median, and 99th-percentile delay and an absolute value of jitter. In contrast to EX_{MQ1}, for EX_{MQ2} we cannot directly compare the results of all flows. Flows 1 and 2 were using the *unlimited* `Iperf3` configuration, while Flows 3 and 4 used the *limited* one. The latter two will be considered as flows corresponding to SR classes B and A respectively in the subsequent experiments. This flow configuration means that we inherently expect different values between the two flow groups. Therefore, in EX_{MQ2} and subsequent experiments involving CBS qdisc we focus on the fulfillment of the SR classes A and B requirements.

For Flows 1 and 2 we observe higher delay values than for Flows 3 and 4, with all mean delays in EX_{MQ2} being generally higher than those in EX_{MQ1}. The average delay for *unlimited* flows reaches as high as 3.94 ms for Flow 1, with the 99%ile of 4.8 ms. For *limited* flows the delay is much lower. The highest recorded mean is 1.81 ms, with the largest 99%ile of 2.19 ms. Again, the median delay closely follows the average in most experiments. The recorded delay values in EX_{MQ2} indicate that on average the MQPRIO qdisc alone can fulfill the requirements of SR class A for *limited* flows, however the 99%ile values exceed the required delay of 2 ms. Both Flows 3 and 4 fulfill the requirement of SR class B.

With the absolute values of jitter in EX_{MQ2} we observe somewhat lower mean values for *unlimited* flows compared to the *limited* ones. However, the median absolute jitter is comparable across all four flows. While the mean and median absolute jitter falls within the requirements of SR class A, the 99%ile values are far higher than the 125 μs upper bound. On the contrary, the highest 99%ile absolute jitter of 778.20 μs falls within the requirements of SR class B.

To summarize, in EX_{MQ1} we did not observe any clear patterns that would indicate that any of the NIC queues are prioritized. Furthermore, we notice that while MQPRIO qdisc by itself can support SR class A requirements on average, to achieve the 125 μs requirement for the 99%ile absolute jitter values, some optimization using TSN qdiscs is required. Based on EX_{MQ2} we come to a similar conclusion, justifying the need for experiments whose results are presented in the subsequent sections.

2) CBS SINGLE FLOW VERIFICATION

Following the experiments involving MQPRIO, we investigate the fulfillment of SR class A requirements with a single flow policed by the CBS qdisc in EX_{CS} experiments as introduced in Section VI-A2. The measurements are presented in Figure 22 showing the delay and jitter values corresponding

TABLE 10. Experiment EX_{MQ2} mean ($\bar{x}$), median ($\tilde{x}$), and 99th-percentile (99%ile) delay and absolute value of jitter for each flow.

Hops	Flow 1			Flow 2			Flow 3			Flow 4		
	$\bar{x}$	$\tilde{x}$	99%ile	$\bar{x}$	$\tilde{x}$	99%ile	$\bar{x}$	$\tilde{x}$	99%ile	$\bar{x}$	$\tilde{x}$	99%ile
Delay												
1	0.63	0.63	0.80	0.67	0.66	1.09	0.22	0.22	0.37	0.22	0.22	0.37
2	1.57	1.53	2.23	1.04	1.05	1.27	0.56	0.57	0.76	0.53	0.53	0.75
3	2.40	2.43	3.50	1.48	1.46	1.81	0.77	0.78	1.01	0.77	0.78	0.98
4	3.12	3.27	3.75	1.95	1.87	2.48	1.09	1.11	1.38	1.10	1.11	1.37
5	2.91	2.91	3.29	2.61	2.68	3.05	1.31	1.32	1.63	1.33	1.35	1.59
6	3.83	3.92	4.60	2.86	2.78	3.43	1.53	1.55	1.86	1.53	1.54	1.84
7	3.94	3.95	4.80	3.38	3.41	3.73	1.80	1.82	2.19	1.81	1.84	2.12
Jitter												
1	48.44	32.66	164.75	48.44	32.66	165.46	47.75	28.37	262.74	48.51	28.37	262.74
2	48.30	23.60	165.94	48.37	23.60	166.89	75.38	38.39	442.98	73.97	38.39	439.64
3	48.25	32.66	165.22	48.17	32.42	164.75	79.82	28.61	489.95	71.00	28.61	449.18
4	48.44	23.60	166.89	48.04	23.60	166.65	100.98	38.39	626.33	99.65	38.39	615.36
5	48.30	32.42	166.42	48.29	32.42	165.70	104.98	38.39	662.09	96.64	36.72	616.07
6	48.50	32.66	166.18	48.43	32.66	166.18	105.28	29.09	664.95	102.82	37.19	652.31
7	48.30	32.42	165.70	48.25	32.42	165.94	124.13	38.62	778.20	109.92	38.15	711.92

to the varied number of hops. The red dashed line indicates the 2 ms delay or the 125 μs jitter requirement of SR class A. We distinguish between two types of experiments, EX_{CS1} where CBS is employed only on the source node and EX_{CS2} where the qdisc is applied on all network interfaces.

We observe that in both experiment types, the delay shown in Figure 22a is comparable across the varying number of hops. Furthermore, the average delay increases close to linearly, with each hop adding approximately 0.2 ms. The highest average delay in both EX_{CS1} and EX_{CS2} is 1.43 ms for a network with 7 hops. Based on Figure 22a we can also state that with one *limited* and CBS policed flow in the network, we are able to fulfill the SR class A requirements of a maximum 2 ms in terms of delay over a 7 hop network.

The jitter for both CBS single flow experiments is shown in Figure 22b. We again distinguish between the two types of performed experiments with EX_{CS1} and EX_{CS2}. The plot shows the jitter in μs on a logarithmic scale depending on the number of hops used in the network. The red dashed lines indicate the 125 μs allowed jitter for SR class A. For EX_{CS1} we see that the 1%ile and 99%ile of the measured jitter values fall within the 125 μs target across all hop number combinations. However, in experiments involving two to seven hops we observe outliers that exceed this requirement. Furthermore, in each experiment, we measure a mean jitter of 0 μs and observe a median jitter of 0 μs (not shown in the plot), indicating that there is no clock drift in the system. We can also conclude that the average delay stays constant over time. In EX_{CS2} we see similar outcomes as in EX_{CS1}, the only difference being a smaller 1%ile to 99%ile jitter range for experiment with two hops.

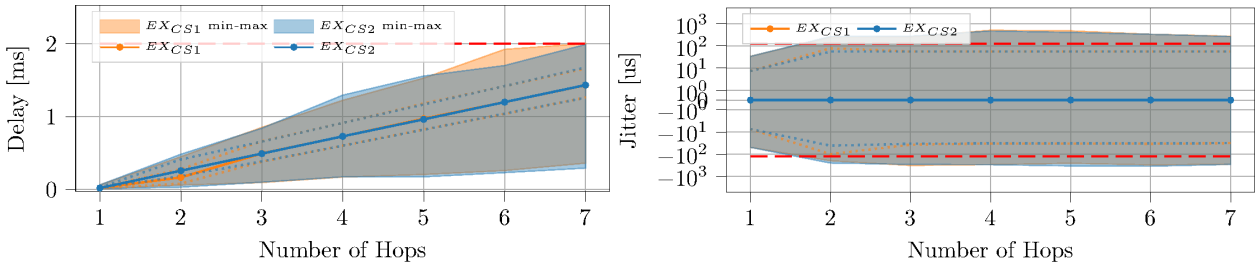
These results show that with one flow, the addition of the CBS qdisc on each interface has little impact on the observed delay and jitter. This does not imply that the qdiscs should not be added on all network interfaces, as with competing traffic, the flows could still be affected. The outcomes of EX_{CS1} and EX_{CS2} indicate that the application of CBS on every node in the investigated network does not negatively impact the performance of the system.

3) CBS MULTIPLE FLOW VERIFICATION

As a next step in validating the experimental environment, we perform the experiments verifying the coexistence of traffic of SR classes A and B with best effort traffic as outlined in Section VI-A3. In this EX_{CM} scenario with a fully loaded network, we focus on the delay and jitter measured for the two *limited* `Iperf3` flows configured for PCPs 3 and 2 corresponding to SR classes A and B respectively. Similarly, as in Single Flow CBS Verification, we run two types of experiments, EX_{CM1} where the CBS qdiscs are configured only on the source node and EX_{CM2} where the qdiscs are configured on each interface in the network. We evaluate the results in two parts, initially focusing on the higher priority flow of SR class A shown in Figure 23, later comparing it against the flow of SR class B presented in Figure 24.

We start with evaluating the delay of SR class A `Iperf3` flow shown in Figure 23a. In scenario EX_{CM1} we see the mean delay rising linearly with the increasing number of hops. The maximum delay reaches 2 ms similarly as in single flow scenarios EX_{CS} with the average increasing from 0.11 ms for 1 hop up to 1.54 ms for 7 hops. Each hop increases the average delay by 0.24 ms. With CBS applied on all interfaces in EX_{CM2}, we observe a slightly higher delay compared to EX_{CM1}. While the latency for one hop remains the same, each additional hop increases the delay by 0.3 ms with the maximum measured of 2.42 ms for seven hops. The mean latency in EX_{CM2} increases from 0.11 ms with 1 hop to 1.90 ms for 7 hops. While the average latency fulfills the requirements of SR class A, the maximum delay exceeds the 2 ms requirement denoted by the red dashed line in the plots.

To better understand the functionality of CBS for SR class A `Iperf3` flow in EX_{CM} we also look at the jitter visualized in Figure 23b. For both EX_{CM1} we measured an average jitter of 0 μs indicating that the delay is not monotonically



(a) Delay (b) Jitter

FIGURE 22. Measured mean delay and jitter for EX_{CS} verification experiments. Colored area represents the range between minimum and maximum values. The dotted lines represent the range between the 1%ile and the 99%ile. The red dashed line represents the target for SR class A.

increasing or decreasing during the experiment run. While for each number of hops, most measured jitter values stay within the 125 μs requirement, we observe outliers exceeding the target both in the positive and negative value range. The high jitter shown for EX_{CM1} amounts to roughly 5% of all measured values. The largest jitter of -952.48 μs and 613.21 μs occurs in experiment with 6 hops. In EX_{CM2} we observe generally lower values compared to EX_{CM1}. Again, the mean jitter is 0 μs indicating that the average latency does not vary over time. We still measure some outliers; however, only at most 0.085% of, in the case of 5 hop experiment, observed values lie outside of the target -125 μs to 125 μs range, with the 1%ile and 99%ile also being within the aforementioned range.

Similarly to SR class A results, we look at the delay and jitter values for the SR class B `Iperf3` flow in EX_{CM} shown in Figure 24. The EX_{CM1} delay of SR class B flow presented in Figure 24a sees comparable delay to that of the higher priority class A flow in EX_{CM1}. Of note, the plot does not show the target maximum delay for SR class B of 10 ms which is outside of the visualized value range. Again, we observe a linear increase in delay with each hop adding 0.24 ms. The measured values do not exceed 2 ms in the 7 hop experiment, well within the 10 ms SR class B requirement. The flow of SR class B experiences a slightly higher latency in EX_{CM2} with the mean increasing linearly from 0.11 ms to 1.9 ms, each hop increasing it by 0.3 ms. The maximum measured delay for seven hops is 2.41 ms.

Lastly, we look at the jitter of SR class B flow shown in Figure 24b. In EX_{CM1}, similarly to previous jitter measurements, we observe a mean value of 0 μs. The same observation can be made for the jitter in EX_{CM2}. In both types of experiments and across all possible numbers of hops, the SR class B jitter requirement of maximum 1000 μs, indicated in the plot by the red dashed line, is fulfilled. Of note, the application of CBS qdisc on all network interfaces in EX_{CM2} generally lowers the jitter compared to when CBS is only configured on the source node in EX_{CM1}.

4) TAPRIO VALIDATION

Starting with the validation of the TAPRIO along with ETF, we introduce a first of results for the EX_{TS} single flows either categorized as SR class A or B and running a deadline or strict ETF modes. Table 11 shows a comparison of the two modes for one to seven hops. The parameters for this results are `txtime-delay`=200 μs and `delta`=175 μs, and windows are spaced as shown in Figure 20. Besides, we show `txtime-delay`=450 μs with `delta`=175 μs, to show the impact of higher `txtime-delay`. For brevity, we only show the results for seven hops. Starting with the deadline mode, we see approximately 200 μs increase based on the 99th percentile results over each hop. Looking at the absolute jitter, we see an increase with each hop where the mean and median are around 130 μs for the seven hops, but when observing the 99th percentile, we see much higher values of up to 737.9055 μs The behavior for the SR class A and B are comparable as well as for the `txtime-delay` of 450 μs.

When comparing the two modes, we observe that ETF strict mode has high difference of mean and median values ranging from 5.08 to 183.23 μs and 4.53 to 31.70 μs respectively. Looking at the case of `txtime-delay`=450 μs with `delta`=175 μs seven hops 99th percentile values, we observe less than 11.5 μs when comparing the SR class A and B results. On the other hand, when looking at the delay, we observe an increase of roughly 1 ms per hop, which corresponds to the window cycle. We observe a lower delay for the lower `txtime-delay` but higher absolute jitter. On the other hand, the higher value results in packets waiting for approximately one cycle before it is forwarded to the next hop; thus, the absolute jitter gets lower.

Figure 25 shows an example configuration, where for each hop, we account Δ, which is the delay it takes to process the packet on each hop. Such setup is evaluated in the EX_{TM}. Using the shift on each hop, once the packet reaches the next hop, a corresponding priority window will be opened to process the packet immediately, without waiting for a whole window cycle. Therefore, it lowers the end-to-end delay in the case of strict traffic. On the other hand, the jitter might fluctuate if the Δ is not properly configured, as some packets might be forwarded right away while others need to wait.

Based on the results of the deadline mode, we see approximately 150 μs overhead caused by each hop, so we set Δ to 200 μs. Worthy of a note, finding an optimal value for this

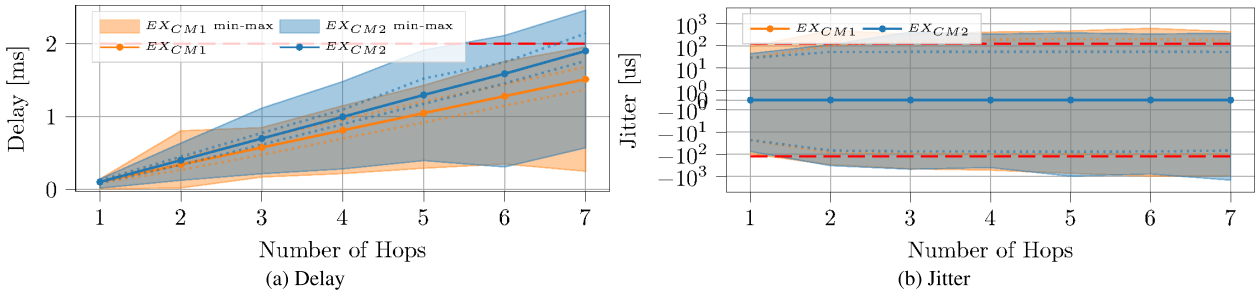


FIGURE 23. Measured mean delay and jitter in EX_{CM} verification experiments for the high priority SR class A flow. Colored area represents the range between minimum and maximum values. The dotted lines represent the range between the 1%ile and the 99%ile. The red dashed line represents the target for SR class A.

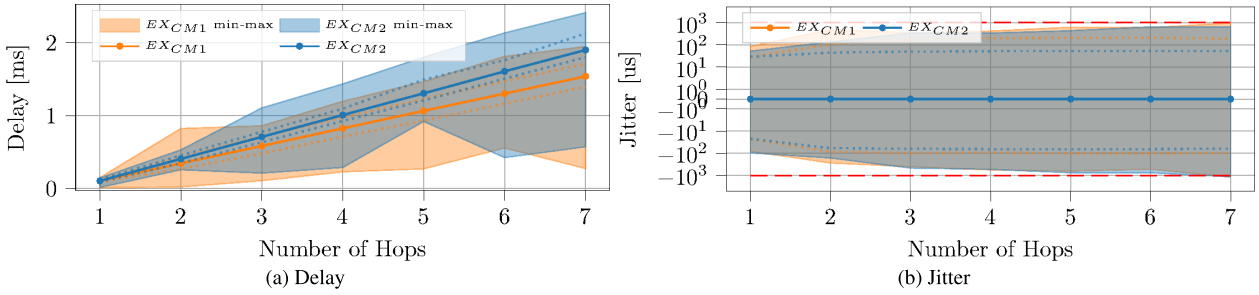


FIGURE 24. Measured mean delay and jitter in EX_{CM} verification experiments for the lower priority SR class B flow. Colored area represents the range between minimum and maximum values. The dotted lines represent the range between the 1%ile and the 99%ile. The red dashed line represents the target for SR class B.

setup is challenging, and to the best of our knowledge, no dedicated algorithm can be used to find the optimal values for the COTS hardware. As identified in the related work, network calculus approaches can provide upper bounds, but those models are directly applicable to a hardware setup, as there are setup-dependent parameters that are not considered.

Table 12 shows three flows, corresponding to SR class A and B and BE, where on each hop is added a $\Delta = 200$ μs corresponding to delay of each hop. In addition, EX_{TM} evaluates the impact of offload and no-offload, as the results in Section V-D provide non-inclusive results that are worthy of evaluation over multiple hops. We adjusted the window sizes so the SR class A window is 400 μs, SR class B is 300 μs and BE is 300 μs. With the 1 Gbit links and 1250B packets (including all headers), we have a transmission time of 100 μs, which should fit three to four packets, depending on the window size. In comparison to EX_{TS}, we also lowered the `txtime-delay` value to 180 μs.

Starting with the behavior of ETF in deadline mode and BE, we see many similarities to delay and absolute jitter. The delay 99th-percentile over seven hops is for both around1.44 ms and the absolute jitter is 708.1 and 469.21 μs for the deadline mode and BE accordingly. We see an improvement with no-offload in place for both setups, as the delay and absolute jitter values get lower. Nevertheless, the values for deadline and BE still fluctuate, which is to be expected due to the nature of their operation. However, even with that in mind, we see SR class B requirements can be fulfilled using the ETF in deadline mode. When comparing the results to EX_{TS}, we see lower values that might be caused

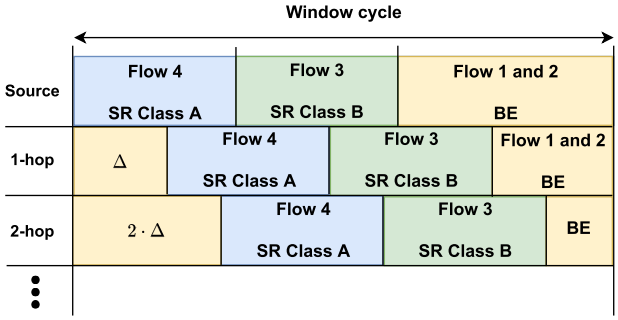


FIGURE 25. Example of TAPRIO window shift over hops and planned child qdiscs configuration with respect to SR classes.

partially by the lower `txtime-delay` value and window shift.

Next, the ETF in strict mode shows delay mean from 0.17 to 3.09 μs with offload and 0.03 to 1.19 μs. Here we see the impact of the shift in comparison to the naive approach in EX_{TS} along with the lower `txtime-delay`. Absolute jitter mean and median improves significantly with no-offload resulting in mean values of 112.97 μs and median of 12.40 μs. On the other hand, the 99th percentile does not show a significant improvement in the no-offload values. Overall, looking at the ETF strict results, we see that with no-offload SR class A requirements can be fulfilled when considering mean and median, but it is not achieved for all values. Nevertheless, aligning the windows over each hop shows an improvement in the results.

Using the EX_{TM} outcome and analysis, we observe that the delay and absolute jitter mean and median show expected

TABLE 11. EX_{TS} results - TAPRIO with ETF in deadline and strict mode for one to seven hops and a single flow, `txtime-delay`=200 and 450μs and ETF `delta`=175μs with offload. Shows delay and absolute jitter mean ($\bar{x}$), median ($\tilde{x}$), and 99th-percentile (99%ile) for increasing number of hops.

Hop Count	ETF Deadline Delay [ms] $\bar{x}$	$\tilde{x}$	99%ile	ETF Deadline Abs Jitter [µs] $\bar{x}$	$\tilde{x}$	99%ile	ETF Strict Delay [ms] $\bar{x}$	$\tilde{x}$	99%ile	ETF Strict Abs Jitter [µs] $\bar{x}$	$\tilde{x}$	99%ile
Configured on HW Queue 1, SR Class A - txtime-delay = 200us												
1	0.0146	0.0131	0.0260	2.4025	0.2384	12.1593	0.1666	0.1678	0.1748	5.0803	4.5300	19.3119
2	0.1723	0.1862	0.2301	28.1509	5.7220	144.0048	0.9869	1.0195	1.1537	95.8912	6.6757	761.0321
3	0.3324	0.3557	0.4139	59.4127	14.5435	273.7045	1.8884	2.1284	2.1536	183.2371	5.9605	930.3093
4	0.4934	0.5732	0.6204	82.5009	10.9673	389.3375	2.6821	2.9578	3.1140	173.2732	31.7097	764.1315
5	0.6366	0.6876	0.8142	108.0719	78.9165	534.5345	3.5595	3.3514	4.0107	168.1013	8.3447	764.3700
6	0.7690	0.8631	1.0040	123.0601	125.1698	607.0137	4.4448	4.1935	4.9787	173.8861	8.3447	764.6084
7	0.8834	0.9284	1.3261	139.7712	128.2692	737.9055	5.3293	5.1799	5.9557	156.7665	7.3910	762.2242
Configured on HW Queue 1, SR Class A - txtime-delay = 450us												
7	0.9150	0.9732	1.2629	130.7446	129.2229	739.8129	6.1695	6.1707	6.1741	3.4798	3.3379	11.4441
Configured on HW Queue 2, SR Class B - txtime-delay = 200us												
1	0.0148	0.0131	0.0279	3.0300	0.2384	14.7820	0.1695	0.1709	0.1743	3.2907	3.0994	13.5899
2	0.1753	0.1905	0.2205	33.7225	6.4373	142.5743	0.9734	1.0474	1.1585	128.9811	9.2983	749.3496
3	0.3414	0.3905	0.4303	53.7798	5.4836	241.0412	1.8174	2.1310	2.1436	182.0896	2.6226	912.6663
4	0.4927	0.5682	0.6189	80.8059	5.2452	385.2844	2.6924	2.4068	3.1335	181.9570	2.1458	912.6663
5	0.6204	0.6676	0.8237	110.7659	16.2125	559.3300	3.5133	3.2718	4.0784	173.6136	33.8554	800.8480
6	0.7886	0.8664	1.1411	124.4057	128.2692	682.8690	4.4417	4.1990	5.1095	183.4610	5.722	923.8720
7	0.8954	0.9344	1.3337	140.9588	129.6997	824.6899	5.3158	5.1868	6.1011	156.2842	3.8147	910.0437
Configured on HW Queue 2, SR Class B - txtime-delay = 450us												
7	0.8960	0.9637	1.2982	143.2839	127.7924	791.5497	6.1697	6.1707	6.1743	3.4249	3.3379	10.7288

values with an increase of delay 200 μs per hop and bounded jitter. However, the 99th percentile shows much higher values. The experiments demonstrate the artifacts caused by a misalignment of the window cycle time and traffic generation. Therefore, EX_{TS-T} increases the generated inter-frame packet spacing from 100 μs to 1 ms. On the other hand, EX_{TS-W} lowers the window cycle to 100 μs, so we generate traffic with 100 μs inter-frame packet spacing. This also means the parameter Δ is lowered to 20 μs. To note, for both EX_{TS-T} and EX_{TS-W}, we consider only the seven hops results for brevity reasons.

Figure 26 shows results for EX_{TS-T} and EX_{TS-W} with and without offload. The EX_{TS-T} delay performs better in comparison to the EX_{TS-W}, where mean and median reach below 1.5 ms with and without offload. This behavior is similar to the jitter, where EX_{TS-T} reaches values below 12 μs for mean and median. EX_{TS-W} shows similar behavior as the EX_{TS} for jitter, which makes us believe the parameter Δ is set too low in this case. Unfortunately, for all scenarios, we see (not only) outliers above 2 ms for the delay.

Figure 26a presents both experiment results with and without offload. Overall, the offload performs better with lower spread and lower delay. Concerning jitter, Figure 26b shows that the majority of values for EX_{TS-T} are within bounds SR class A bounds. In contrast, values observed in EX_{TS-W} experiments fluctuate more significantly.

Overall, EX_{TS-T} and EX_{TS-W} show behavior for traffic patterns and their window sizes. Using higher inter-frame spacing for packets allows for higher window cycles. After all experiments, we can conclude that offload behaves worse than no offload. This behavior might be specific to the use of Intel® I210 and would require investigation on other NICs.

B. EXEMPLARY USE-CASE

In the following we showcase the results for the exemplary use-case investigating the coexistence of traffic policed for SR classes A and B with other types of traffic, mainly BE. The experiments have been conducted using synthetically generated, realistic traffic patterns as outlined in Table 7 and investigated two policing configuration variants, one including the CBS and the other the TAPRIO qdiscs.

1) CBS VARIANT

First, we consider the CBS variant of the experiments and investigate the delay measured in the network for each of the flow paths. We focus only on the policed traffic of flow paths 1, 2, and 3, showing the delays for each of the individual traffic types along the path. The resulting measurements are presented in Figure 27. Starting with scenario EX_{UCC} including all five flow paths in the experiment shown in Figure 27a, we observe that within all flow paths and traffic types we can fulfill the respective requirements of SR classes A and B. We also observe the lowest average delay for the shortest flow path 1, with the mean increasing close to linearly with the increasing number of hops for flow paths 3 and 2, respectively. Furthermore, we observe that the delays for each of the traffic types within a class are similar.

In scenario EX_{UCC-F} presented in Figure 27b, we observe similar or lower delay compared to EX_{UCC} for each of the flow paths. Again, the delays along a flow path across the various traffic types are similar. We further see that the minimum

TABLE 12. EX_{TM} **results -TAPRIO with ETF for one to seven hops for three flows corresponding to SR class A, B, and BE.** `txtime-delay`**=180 and ETF** `delta`**=175** μ**s with offload and** $\Delta = 200$**, delay and absolute jitter mean (**$\bar{x}$**), median (**$\tilde{x}$**), and 99th-percentile (99%ile) for increasing number of hops.**

Hops	Strict mode $\bar{x}$	Strict mode $\tilde{x}$	Strict mode 99%ile	Deadline mode $\bar{x}$	Deadline mode $\tilde{x}$	Deadline mode 99%ile	Best Effort $\bar{x}$	Best Effort $\tilde{x}$	Best Effort 99%ile
				Offload					
Delay									
1	0.1697	0.1709	0.1740	0.0145	0.0124	0.0255	0.0292	0.0274	0.0687
2	0.6230	0.5047	1.1215	0.2285	0.2053	0.3927	0.2008	0.2148	0.2687
3	1.0018	0.8843	1.5223	0.3754	0.3996	0.5240	0.3889	0.4191	0.0005
4	1.5102	1.7383	1.8868	0.4786	0.4718	0.8163	0.6574	0.6375	1.0378
5	2.0067	2.0828	2.8648	0.7464	0.7427	0.9587	0.7946	0.7503	1.2007
6	2.7753	2.9056	3.8459	0.9268	0.9463	1.1604	0.9756	0.9682	1.3514
7	3.0854	3.3307	3.5398	0.9915	0.9890	1.4389	1.1128	1.1239	1.4248
Jitter									
1	3.6581	3.8147	10.7288	2.7352	0.2384	13.3514	6.7619	3.5763	49.1142
2	140.6235	15.2588	737.4287	35.8777	4.0531	285.8639	28.8236	3.8147	169.754
3	133.5313	3.5763	619.4115	36.0235	4.0531	264.6446	41.3678	5.0068	237.7033
4	159.1760	3.5763	910.0437	79.0190	10.0136	457.6302	34.7197	4.2915	479.4598
5	242.2621	42.2001	944.1376	51.3121	3.8147	374.3172	86.7405	5.0068	558.9557
6	234.5140	79.8702	976.8009	52.7781	3.5763	469.6846	95.223	4.53	650.8827
7	272.4967	37.1933	957.6797	101.9346	19.0735	708.1032	65.0604	5.9605	469.2078
				No-Offload					
Delay									
1	0.0309	0.0315	0.0513	0.0198	0.0124	0.0587	0.0189	0.0129	0.0575
2	0.2316	0.2310	0.2508	0.1877	0.1876	0.2632	0.1852	0.1996	0.2522
3	0.3616	0.3598	1.0846	0.3460	0.3543	0.4373	0.3373	0.3467	0.4370
4	0.6840	0.6320	1.2951	0.5069	0.5245	0.6585	0.5109	0.5264	0.7217
5	0.8421	0.8194	1.0111	0.6997	0.7253	0.8376	0.6493	0.7246	0.8466
6	1.0085	0.9427	1.8860	0.8478	0.8867	1.0431	0.8415	0.8490	1.0900
7	1.1928	1.1230	2.1133	0.9777	1.0068	1.1854	0.9772	0.9816	1.2145
Jitter									
1	4.7969	4.05310	32.4249	1.7761	0.2384	20.9808	3.8164	0.4768	30.7560
2	2.7334	0.9537	18.3582	16.0992	3.8147	141.8591	23.7676	2.6226	157.1178
3	42.0803	11.6825	756.0253	32.3436	4.0531	223.9823	43.1111	3.3379	233.4118
4	122.4446	11.4441	690.5913	50.2742	4.0531	315.4278	51.3208	2.6226	354.8765
5	45.0614	3.5763	207.4242	46.4492	5.4836	490.427	97.4545	4.7684	564.3368
6	122.8868	12.3978	959.1579	71.6073	3.8147	558.3763	47.4129	5.0068	356.6742
7	112.9722	12.3978	950.3365	70.4429	9.7752	478.9829	52.5589	4.7684	367.8799

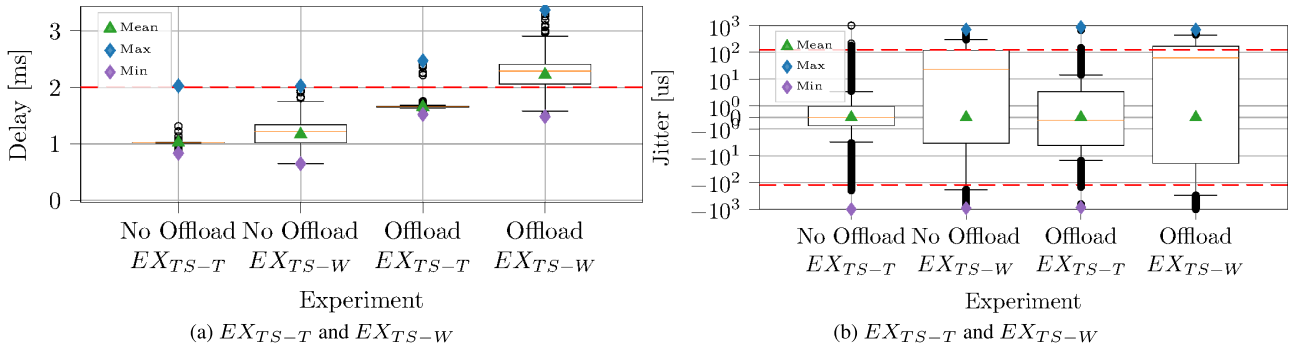
delay is directly correlated with the path length. It increases linearly with the increasing length of the path. The mean delay shows a slightly different trend compared to EX_{UCC}. Due to the high number of outliers we do not observe a linear increase in the average delay with the increasing number of hops. This is caused by the various volumes of traffic in each experiment and the absence of competing traffic in the network. Flow path 1 supports the highest bitrate in EX_{UCC-F} of 89.152 Mbit/s, hence we observe a wider spread of the measured delays which is comparable with EX_{UCC}. Flow paths 2 and 3 with total bitrates of 26.4 Mbit/s and 12.82 Mbit/s do not experience as much competition within their traffic classes. We observe a significantly lower delay for flows in EX_{UCC-F}, as the flows are placed alone in the network in individual experiments and there is no external competition for resources.

Continuing with the observed jitter presented in Figure 28 we again compare the jitter of EX_{UCC} with that of EX_{UCC-F} shown in Figure 28a and 28b respectively. For EX_{UCC} we observe a similar distribution of jitter values across all flow paths and traffic types. While the SR class B requirements relevant for flow path 3 are fulfilled, the outliers observed for traffic types along flow paths 1 and 2 exceed the corresponding requirements of SR class A. This is expected, as CBS is mostly tailored towards guaranteeing bitrates, and only gives limited bounds on delay or jitter. With that, we still see that the vast majority of the jitter values on flow paths 1 and 2 falls within the 125 μs required by the SR class A. In EX_{UCC-F} we observe a generally lower jitter compared to EX_{UCC} with all traffic types along flow path 2 achieving the requirements of SR class A. With the higher bitrates, flow path 1 sees outliers in the jitter above 125 μs.

To summarize, in the CBS variant of the use-case experiments, we observe that the introduction of cross-traffic and more complex flows negatively impacts the delay and jitter observed in the network. Yet, the observed values in most cases fall within the requirements of SR class A and completely fulfill the requirements of SR class B.

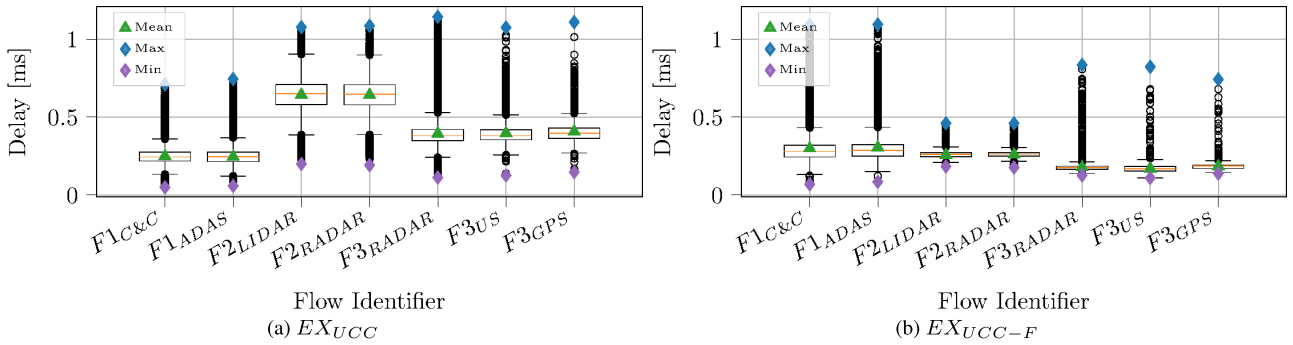
2) TAPRIO VARIANT

The exemplary use-case with TAPRIO showcases the results coexistence of policed traffic for SR classes A and B with BE. As for the CBS variant, we rely on the synthetically



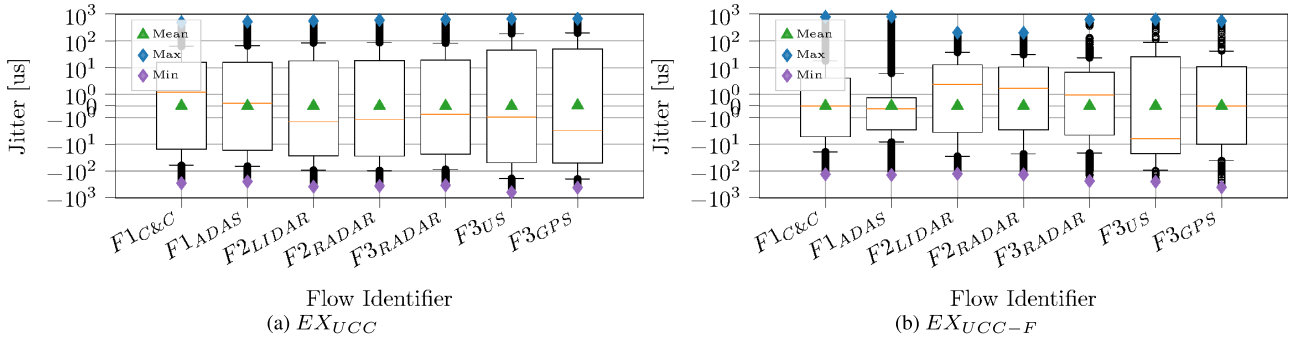
(a) EX_{TS-T} and EX_{TS-W} (b) EX_{TS-T} and EX_{TS-W}

FIGURE 26. Boxplots of measured delay and jitter for the TAPRIO EX_{TS-T} and EX_{TS-W} with and without offload.



(a) EX_{UCC} (b) EX_{UCC-F}

FIGURE 27. Boxplot of measured delay for the exemplary use-case in the CBS variant.



(a) EX_{UCC} (b) EX_{UCC-F}

FIGURE 28. Boxplot of measured jitter for the exemplary use-case in the CBS variant.

generated traffic based on realistic traffic patterns as outlined in Table 7. For the TAPRIO we investigate the difference of using ETF in deadline and strict mode for the SR classes with (EX_{UCT}) and without cross traffic (EX_{UCT-F}). We used the same length of experiments and payload sizes and re-used the default window sizes values as shown in Figure 20. For all of the experiments we used `txtime-delay`=450 μs and ETF `delta`=175 μs as with these values we achieved lower absolute jitter as shown in EX_{TS}. We do not aim to find optimal results as shown for EX_{TS-T} and EX_{TS-W}. The main focus of these experiments is on cross-traffic behavior. Ideally, we would try to find window cycles respecting the lowest packet inter-frame spacing based on the Table 7. Also, we do not show results of the unpoliced traffic for brevity reasons, as it is irrelevant compared to higher priority traffic.

Table 13 shows all of the results for policed traffic corresponding to SR class A and B. Starting with the SR class A traffic that is presented in Flow 1 and 2, we observe maximum delay values of the 99th-percentile for EX_{UCT-F} of 1.17 μs for Flow 1 and 3.17 μs. This corresponds to previous results when no cross traffic is introduced and we observe roughly one window cycle increase per hop. However, due to the traffic patterns sharing a single window, we observe higher absolute jitter with maximum values for the 99th-percentile of 197.41 μs for Flow 1 and 257.73 μs for Flow 2. This is much worse performance than previous absolute jitter values, and it shows that single flows when sharing the window compete with the same hardware queue and require window size adjustments. This increase of absolute jitter is even more prominent when continuing with EX_{UCT} for Flows 1 and 2, resulting in values higher than 800 μs. For Flow 1, we can

TABLE 13. Exemplary use-case for TAPRIO with ETF for Flow 1, 2, and 3 for corresponding SR class A and B flows. `txtime-delay`=450 μs and ETF `delta`=175 μs with offload, delay and absolute jitter mean ($\bar{x}$), median ($\tilde{x}$), and 99th-percentile (99%ile) with (EX_{UCT}) and without cross-traffic (EX_{UCT-F}).

	ETF Deadline						ETF Strict					
Flow	Delay [ms]			Abs Jitter [µs]			Delay [ms]			Abs Jitter [µs]		
Identifier	$\bar{x}$	$\tilde{x}$	99%ile	$\bar{x}$	$\tilde{x}$	99%ile	$\bar{x}$	$\tilde{x}$	99%ile	$\bar{x}$	$\tilde{x}$	99%ile
Without cross-traffic Flow Path 1 - EX_{UCT-F}												
$\mathbf{F1_{C\&C}}$	0.1948	0.2146	0.2418	51.9020	10.7288	167.1314	1.0187	1.0033	1.1683	24.8401	3.8147	197.4106
$\mathbf{F1_{ADAS}}$	0.1190	0.1311	0.1891	40.0371	53.1673	111.5799	1.1644	1.1668	1.1735	7.0455	6.1989	24.0803
With cross-traffic Flow Path 1 - EX_{UCT}												
$\mathbf{F1_{C\&C}}$	0.1738	0.1795	0.2582	22.7314	4.5300	143.0511	1.9133	1.9655	2.1639	105.8464	3.5763	846.8628
$\mathbf{F1_{ADAS}}$	0.1345	0.1359	0.2401	58.5694	60.3199	168.0851	1.8768	2.1496	2.1651	304.5364	9.7752	853.5385
Without cross-traffic Flow Path 2 - EX_{UCT-F}												
$\mathbf{F2_{LIDAR}}$	0.2655	0.2382	0.4106	55.2973	2.3842	169.9924	3.1633	3.1657	3.1710	6.9258	3.0994	22.1729
$\mathbf{F2_{RADAR}}$	0.2517	0.2289	0.3781	46.7465	2.6226	144.2432	3.0059	2.9223	3.1700	118.0142	28.7294	257.7305
With cross-traffic Flow Path 2 - EX_{UCT}												
$\mathbf{F2_{LIDAR}}$	0.4250	0.4132	0.6468	69.1725	51.7368	297.5464	2.5435	2.4014	3.1722	253.2613	50.5447	849.0086
$\mathbf{F2_{RADAR}}$	0.3156	0.3033	0.5744	88.3602	74.8634	315.1894	2.7698	2.4450	3.1731	319.641	31.9481	822.3057
Without cross-traffic Flow Path 3 - EX_{UCT-F}												
$\mathbf{F3_{RADAR}}$	0.2038	0.1996	0.2284	4.1648	2.6226	27.4181	1.9565	1.9045	2.1689	76.3199	8.8215	274.6582
$\mathbf{F3_{US}}$	0.1967	0.1929	0.2275	6.9516	3.3379	38.3854	1.9163	1.9059	1.9482	20.0547	20.2656	255.8231
$\mathbf{F3_{GPS}}$	0.1785	0.1736	0.2007	4.9583	1.4305	29.9525	1.9195	1.9176	1.9398	11.6139	4.5300	59.5999
With cross-traffic Flow Path 3 EX_{UCT}												
$\mathbf{F3_{RADAR}}$	0.1439	0.118	0.2814	52.9357	9.5367	162.9281	1.9816	1.9817	1.9972	7.3666	5.4836	40.7696
$\mathbf{F3_{US}}$	0.1043	0.1023	0.1299	4.4981	1.4305	37.5700	2.1022	2.1145	2.1276	27.2254	15.0204	151.5794
$\mathbf{F3_{GPS}}$	0.0979	0.0956	0.1237	4.6743	1.4305	33.0472	2.1122	2.1219	2.1362	30.9464	11.2057	152.5164

even observe an increase of maximum delay of the 99th-percentile from 1.17 μs to 2.17 μs, which is an increase of one whole window cycle. This is an interesting outcome, as the competing traffic is in this scenario SR class B traffic. Flow 2 stays the same as the cross traffic is mainly BE traffic, which does not affect the higher priority class.

When observing the results for Flows 1 and 2 using the ETF qdisc in deadline mode, we do not observe similar behavior as with the strict mode. The delay and absolute jitter for Flow 1 with EX_{UCT} and EX_{UCT-F} show similar results without a clear tendency. The results sometimes even showed better performance for EX_{UCT}.

The last part to compare is Flow 3, which corresponds to SR class B. When comparing the EX_{UCT} and EX_{UCT-F} for the strict mode, we see an improvement of absolute jitter values from the previous maximum 99th-percentile of 274.66 μs to 152.52 μs. For the delay is the tendency to be somewhat similar and can be an artifact of a given experiment run. On the other hand, for the deadline mode we can observe worse behavior of the maximum 99th-percentile from 38.39 μs to 162.93 μs. However, for the delay, we see lower values for delay for EX_{UCT}. The explanation for this behavior might be the functionality of the Intel® I210 NIC in offload mode, as once more than a single hardware queue is used (as in the case of EX_{UCT-F}) it enters different internal algorithm how to process the traffic [62].

Overall, we see then even if we configure the SR classes accordingly; the requirements cannot be easily fulfilled when using TAPRIO without identifying optimal parameters for the given setups as introduced in EX_{TS}.

C. REQUIREMENTS FULFILLMENT

With the experiments outlined in Section VI and results shown in Sections VII-A and VII-B we showcased how an TSN experimental environment can be prepared, validated, and used to perform experiments in the time-sensitive and in-vehicular domains. The experiments were supported by the methodology introduced with Section IV as well as various system optimizations, traffic generation and qdisc parameter optimization outlined in Section V. We performed the validation tests and experiments involving an exemplary use-case using our *EnGINE* deployment and verified the results against requirements placed upon such networks, amongst others outlined in Table 1. We concentrated on the highly demanding SR classes A and B. These results aim to be an example as to how the introduced methodology can be applied in practice and how its results can be interpreted.

During the run of the experiments, we closely looked at the CBS and TAPRIO qdiscs and their fulfillment of the aforementioned requirements in both synthetic and more realistic scenarios. The results and the capabilities of supporting the SR classes are summarized in Table 14. In most of the experiments, we fulfilled the SR class requirements in terms of average and median delay and jitter. However, maximum (and minimum in case of jitter) values were sometimes recorded outside of the required ranges for SR class A.

For TAPRIO using ETF in strict mode (EX_{TS-S} and EX_{TM-S}), we showcased that configuration can fulfill bounded latency or bounded jitter with respect to SR class A. Based on this result, we investigated the behavior of traffic patterns concerning window cycle time in EX_{TS-T}, showing it

TABLE 14. Summary of SR class A and B requirement fulfillment by CBS and TAPRIO qdisc in various configurations.

Experiment	Requirements Fulfillment			
	SR class A		SR class B	
	Delay	Jitter	Delay	Jitter
MQPRIO				
EX_{MQ1}	YES	YES*	YES	YES
EX_{MQ2}	YES	YES*	YES	YES
CBS				
EX_{CS1}	YES	YES*	YES†	YES†
EX_{CS2}	YES	YES*	YES†	YES†
EX_{CM1}	YES	YES*	YES	YES
EX_{CM2}	YES*	YES*	YES	YES
TAPRIO				
EX_{TS-D}	YES	NO	YES	YES
EX_{TS-S}	NO	YES	YES	YES
EX_{TM-D}	YES	NO	YES	YES
EX_{TM-S}	YES*	NO	YES	YES
EX_{TS-T}	YES*	YES*	YES	YES
EX_{TS-W}	YES*/NO‡	NO	YES	YES

*Mean and median fulfill the requirements
†Test for SR class A, assume same outcome for class B
‡Not fulfilled when using offload

fulfills the SR class A requirements, except outliers. Overall, the results show a possible direction of what parameters can be evaluated and modified for the qdisc configuration to fulfill the defined requirements. As expected, the ETF in deadline mode offers low latency, but shows high jitter, not fitting the SR class requirement. On the other hand, SR class B requirements are easy to fulfill for both deadline and strict modes, even with a more naive approach. Overall, finding optimal parameters for the TAPRIO window sizes is outside this paper's scope. However, we showcased possibilities for configuring the parameters and their impact on the performance.

Of note, these experiments were performed using the *EnGINE* environment, and results obtained on other deployments may vary. The methods applied to obtain these results apply to any deployment with similar capabilities to that of *EnGINE*. As motivated and generalized in Section IV, the outlined methodology is applicable to other TSN domains.

VIII. CHALLENGES & LIMITATIONS

This sections summarizes the challenges we faced with the proposed methodology and limitations of our evaluation and configurations.

A. METHODOLOGY CHALLENGES

We identified several challenges concerning the three individual steps of the methodology.

The **S1** requires a detailed assessment for the given domain to understand the requirements of the environment and map what traffic patterns are present within the network. Definition of requirements might be less challenging, as many of the real-time domain requirements are well standardized and such requirements can also be applied to the TSN domain. On the other hand, the analysis of the traffic patterns does not scale well with more complex networks, where collecting all details might be unfeasible. However, the approach in such scenarios can be to focus only on the relevant high-priority traffic with tight KPIs boundaries, which lowers the scope.

Similarly as for **S2**, the experiment scale might be challenging for the configuration and evaluation options. For example, finding suitable parameters for TAPRIO window sizes is a hard problem. Therefore, more automated solutions, e.g. using novel machine learning techniques, for performance assessment could be suitable.

The **S3** focused on the optimization to fulfill identified KPIs. Within the scope of this paper, we focused on OS and configuration parameter optimizations. However, for other use-cases, this might require even more low-level improvements, e.g., using Real-time OS, hardware accelerators, or a more detailed parameter study.

The introduced methodology brings new challenges during its deployment and execution that might not apply to every TSN domain. Nevertheless, we believe these challenges apply to the execution rather than the required steps.

B. SETUP LIMITATIONS

For the measurement results, we identified several limitations with our setup. First, we rely on the `tcpdump` to capture traffic in egress and ingress directions. Unfortunately, the egress direction supports only Software (SW) timestamping, which is less precise than HW timestamping. This limitation can introduce additional delay and jitter caused by the time the packet spends in the Linux networking stack, which otherwise would not be included if both timestamps were collected by the NIC HW. Nevertheless, even with this challenge, we see that most packets can fulfill given requirements.

Second, even though we rely on PTP, the clocks in the network can deviate over a larger number of hops. We know the clock synchronization precision on a single node, but it is possible that the clock deviates over a larger number of hops. Nevertheless, we believe that the clock drift is negligible within our network setup and the number of hops.

Finally, in the domain of IVNs additional TSN standards are used, that are not part of our evaluations. Extending the infrastructure and including additional, for instance, asynchronous, TSN standards could provide valuable insights.

IX. CONCLUSION

In this work, we provide a detailed description of a novel methodology focusing on reproducible experiments in the TSN domain. The experiments are enabled by the underlying infrastructure called *EnGINE* that is built using the COTS hardware and open-source solutions. Despite initial focus on IVNs, during the detailed methodology analysis we provide a generalized approach applicable to any TSN deployment. Especially when investigating the requirements and comparing them among different application domains, we identify many similarities that allow for generalization of our methodology.

The methodology and infrastructure come with some challenges and limitations. Starting with the methodology, it requires a lot of effort in the **S1**, concerning the requirements and traffic patterns mapping. As a part of **S2**, we search for suitable configurations. Depending on the domain and network size, it may be a time consuming process The infrastructure requires updates with respect to the offered TSN standards to evaluate additional domains beyond the IVNs focus. Finding COTS hardware is challenging, but there are new open-source projects offering novel software solutions.

We present a case study focusing on the IVNs, including a sample use-case and evaluation of their requirements. Following the introduced methodology steps, we define requirements and traffic patterns present within the IVN topologies. Furthermore, we provide detailed configuration steps and system optimization techniques to offer determinism in the Linux kernel contributing to **S3**.

We cover detailed results for the system and configuration performance in scenarios over seven hops and for a sample use-case. The results identify that the requirements placed on such networks are partially fulfilled, depending on the specific scenario and hardware. The jitter, low packet loss, and latency, even though bounded, sometimes do not reach SR class A requirements. To note, some of these observations are possibly caused by the used configuration specifics that not always follow the identified traffic patterns and might strain the system. However, we can fulfill other requirements identified during the analysis. Furthermore, we show how *EnGINE* can be used to perform TSN experiments based on an experimental use-case with detailed configuration of individual steps. Overall, the methodology and infrastructure are applicable for reproducible and scalable TSN experiments.

ACKNOWLEDGMENT

The authors would like to thank Max Helm and Sebastian Gallenmüller for the discussions during the course of this research and also would like to thank the anonymous reviewers for their valuable feedback.

REFERENCES

[1] W. Zeng, M. A. S. Khalid, and S. Chowdhury, "In-vehicle networks outlook: Achievements and challenges," *IEEE Commun. Surveys Tuts.*, vol. 18, no. 3, pp. 1552–1571, 3rd Quart., 2016.

[2] S. Tuohy, M. Glavin, C. Hughes, E. Jones, M. Trivedi, and L. Kilmartin, "Intra-vehicle networks: A review," *IEEE Trans. Intell. Transp. Syst.*, vol. 16, no. 2, pp. 534–545, Apr. 2015.

[3] F. Rezabek, M. Bosk, T. Paul, K. Holzinger, S. Gallenmüller, A. Gonzalez, A. Kane, F. Fons, Z. Haigang, G. Carle, and J. Ott, "EnGINE: Developing a flexible research infrastructure for reliable and scalable intra-vehicular TSN Networks," in *Proc. 3rd Int. Workshop High-Precision, Predictable, Low-Latency Netw. (HiPNet 2021)*, İzmir, Turkey, 2021, pp. 530–536. [Online]. Available: https://ieeexplore.ieee.org/document/9615529

[4] D. Pannell, L. Chen, J. Dorr, W. Lo, M. Potts, H. Zinner, and A. Zu. (2019). *Use Cases—IEEE P802.1DG V0.4*. Accessed: Jun. 29, 2022. [Online]. Available: https://www.ieee802.org/1/files/public/docs2019/dg-pannell-automotive-use-cases-0919-v04.pdf

[5] *TSN for Aerospace Onboard Ethernet Communications*, Standard P802.1DP. Accessed: Jul. 14, 2022. [Online]. Available: https://1.ieee802.org/tsn/802-1dp/

[6] *TSN Profile for Industrial Automation*, Standard IEC/IEEE 60802. Accessed: Jul. 14, 2022. [Online]. Available: https://1.ieee802.org/tsn/iec-ieee-60802/

[7] *IEEE Standard for Local and Metropolitan Area Networks—Virtual Bridged Local Area Networks Amendment 12: Forwarding and Queuing Enhancements for Time-Sensitive Streams*, IEEE Standard 802.1Qav-2009 (Amendment to IEEE Std 802.1Q-2005), 2010, pp. 1–72.

[8] *IEEE Standard for Local and Metropolitan Area Networks—Bridges and Bridged Networks—Amendment 25: Enhancements for Scheduled Traffic*, IEEE Standard 802.1Qbv-2015 (Amendment to IEEE Std 802.1Q-2014 as amended by IEEE Std 802.1Qca-2015, IEEE Std 802.1Qcd-2015, and IEEE Std 802.1Q-2014/Cor 1-2015), 2016, pp. 1–57.

[9] *IEEE Standard for Local and Metropolitan Area Networks—Timing and Synchronization for Time-Sensitive Applications*, IEEE Standard 802.1AS-2020, 2020, pp. 1–421.

[10] D. Pannell. (2019). *Automotive Ethernet AVB Functional and Interoperability Specification*. Last accessed: May 10, 2022. [Online]. Available: https://avnu.org/wp-content/uploads/2014/05/Auto-Ethernet-AVB-Func-Interop-Spec_v1.6.pdf

[11] *ISO/IEC/IEEE International Standard—Information Technology—Telecommunications and Information Exchange between Systems—Local and Metropolitan Area Networks—Specific Requirements—Part 1BA: Audio Video Bridging (AVB) Systems*, Standard ISO/IEC/IEEE 8802-1BA, 2016, pp. 1–52.

[12] *IEEE Standard for a Transport Protocol for Time-Sensitive Applications in Bridged Local Area Networks*, IEEE Standard 1722-2016 (Revision of IEEE Standard 1722-2011), 2016, pp. 1–233.

[13] *IEEE Standard for Local and Metropolitan Area Network—Bridges and Bridged Networks*, IEEE Standard 802.1Q-2018, 2018, pp. 1–1993.

[14] *IEEE Standard for a Precision Clock Synchronization Protocol for Networked Measurement and Control Systems*, IEEE Standard 1588-2019, 2020, pp. 1–499.

[15] B. Moussa, M. Debbabi, and C. Assi, "A detection and mitigation model for PTP delay attack in a smart grid substation," in *Proc. IEEE Int. Conf. Smart Grid Commun. (SmartGridComm)*, Nov. 2015, pp. 497–502.

[16] R. Cochran. *LinuxPTP*. Accessed: Jul. 18, 2022. [Online]. Available: https://sourceforge.net/p/linuxptp/code/ci/24220e87fdb7464/tree/

[17] R. Cochran and M. Lichvar. *PHC2SYS(8)*. Accessed: Jul. 16, 2022. [Online]. Available: https://linux.die.net/man/8/phc2sy

[18] W. Weiss, D. J. Heinanen, F. Baker, and J. T. Wroclawski, *Assured Forwarding PHB Group*, RFC 2597, Jun. 1999. [Online]. Available: https://www.rfc-editor.org/info/rfc2597

[19] *TC(8)*. Accessed: Jul. 15, 2022. [Online]. Available: https://linux.die.net/man/8/tc

[20] *Configuring VLAN Interfaces, TSN Documentation Project for Linux*. Accessed: Jul. 13, 2022. [Online]. Available: https://tsn.readthedocs.io/vlan.html

[21] *IEEE Standard for Local and Metropolitan Area Networks—Bridges and Bridged Networks—Amendment 25: Enhancements for Scheduled Traffic*, IEEE Standard 802, 2016, pp. 1–57.

[22] *TAPRIO(8)*. Accessed: Jun. 11, 2022. [Online]. Available: https://man7.org/linux/man-pages/man8/tc-taprio.8.html

[23] *MQPRIO(8)*. Accessed: May 23, 2022. [Online]. Available: https://man7.org/linux/man-pages/man8/tc-mqprio.8.html

[24] S. Gallenmüller, D. Scholz, H. Stubbe, and G. Carle, "The pos framework: A methodology and toolchain for reproducible network experiments," in *Proc. 17th Int. Conf. Emerg. Netw. Exp. Technol. (CoNEXT)*, Munich, Germany, Dec. 2021, pp. 259–266.

[25] C. Mauclair, M. Gutiérrez, J. Migge, and N. Navet, "Do we really need TSN in next-generation helicopters? Insights from a case-study," in *Proc. IEEE/AIAA 40th Digit. Avionics Syst. Conf. (DASC)*, Oct. 2021, pp. 1–7.

[26] P.-J. Chaine, M. Boyer, C. Pagetti, and F. Wartel, "TSN support for quality of service in space," in *Proc. 10th Eur. Congr. Embedded Real Time Softw. Syst. (ERTS)*, Toulouse, France, Jan. 2020, pp. 1–12. [Online]. Available: https://hal.archives-ouvertes.fr/hal-02441327

[27] D. Bruckner, M.-P. Stanica, R. Blair, S. Schriegel, S. Kehrer, M. Seewald, and T. Sauter, "An introduction to OPC UA TSN for industrial communication systems," *Proc. IEEE*, vol. 107, no. 6, pp. 1121–1131, Jun. 2019.

[28] A. Sabry, A. Omar, M. Hammad, and N. Abdelbaki, "AVB/TSN protocols in automotive networking," in *Proc. 15th Int. Conf. Comput. Eng. Syst. (ICCES)*, Dec. 2020, pp. 1–7.

[29] M. Ashjaei, G. Patti, M. Behnam, T. Nolte, G. Alderisi, and L. Lo Bello, "Schedulability analysis of Ethernet audio video bridging networks with scheduled traffic support," *Real-Time Syst.*, vol. 53, no. 4, pp. 526–577, Jul. 2017.

[30] X. Huang, P. Wang, X. Cheng, D. Zhou, Q. Geng, and R. Yang, "The ApolloScape open dataset for autonomous driving and its application," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 42, no. 10, pp. 2702–2719, Oct. 2020.
[31] J. Migge, J. Villanueva, N. Navet, and M. Boyer, "Insights on the performance and configuration of AVB and TSN in automotive Ethernet networks," in *Proc. 9th Eur. Congr. Embedded Real Time Softw. Syst. (ERTS 2018)*, Toulouse, France, Jan. 2018, pp. 1–10.
[32] *Time Sensitive Networks for Flexible Manufacturing Testbed Characterization and Mapping of Converged Traffic Types*, Ind. Internet Consortium, Milford, MA, USA, 2019.
[33] G. Miranda, E. Municio, J. Haxhibeqiri, D. F. Macedo, J. Hoebeke, I. Moerman, and J. M. Marquez-Barja, "Evaluating time-sensitive networking features on open testbeds," in *Proc. IEEE Conf. Comput. Commun. Workshops (INFOCOM WKSHPS)*, May 2022, pp. 1–2.
[34] M. H. Farzaneh and A. Knoll, "Time-sensitive networking (TSN): An experimental setup," in *Proc. IEEE Veh. Netw. Conf. (VNC)*, Nov. 2017, pp. 23–26.
[35] J. Pfrommer, A. Ebner, S. Ravikumar, and B. Karunakaran, "Open source OPC UA PubSub over TSN for realtime industrial communication," in *Proc. IEEE 23rd Int. Conf. Emerg. Technol. Factory Autom. (ETFA)*, Sep. 2018, pp. 1087–1090.
[36] J. Jiang, Y. Li, S. H. Hong, M. Yu, A. Xu, and M. Wei, "A simulation model for time-sensitive networking (TSN) with experimental validation," in *Proc. 24th IEEE Int. Conf. Emerg. Technol. Factory Autom. (ETFA)*, Sep. 2019, pp. 153–160.
[37] P. Didier. (2020). *Testbeds—Time Sensitive Networks for Flexible Manufacturing*. Accessed: Jul. 12, 2022. [Online]. Available: https://hub.iiconsortium.org/time-sensitive-networks
[38] P. Didier and J. Fontaine, "Results, insights and best practices from IIC testbeds: Time-sensitive networking testbed," Ind. Internet Consortium, Milford, MA, USA, Tech. Rep., 2017.
[39] L. Zhao, P. Pop, and S. Steinhorst, "Quantitative performance comparison of various traffic shapers in time-sensitive networking," *CoRR*, vol. abs/2103.13424, pp. 1–27, Jun. 2022.
[40] J. Walrand, M. Turner, and R. Myers, "An architecture for in-vehicle networks," *IEEE Trans. Veh. Technol.*, vol. 70, no. 7, pp. 6335–6342, Jul. 2021.
[41] S. Mubeen, M. Ashjaei, and M. Sjodin, "Holistic modeling of time sensitive networking in component-based vehicular embedded systems," in *Proc. 45th Euromicro Conf. Softw. Eng. Adv. Appl. (SEAA)*, Aug. 2019, pp. 131–139.
[42] T. Steinbach, H. D. Kenfack, F. Korf, and T. Schmidt, "An extension of the OMNeT++ INET framework for simulating real-time Ethernet with high accuracy," in *Proc. 4th Int. ICST Conf. Simulation Tools Techn. (SIMUTools)*. Brussels, Belgium: Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering, 2011, pp. 375–382.
[43] J. Falk, D. Hellmanns, B. Carabelli, N. Nayak, F. Dürr, S. Kehrer, and K. Rothermel, "NeSTiNg: Simulating IEEE time-sensitive networking (TSN) in OMNeT++," in *Proc. Int. Conf. Netw. Syst. (NetSys)*, Mar. 2019, pp. 1–8.
[44] L. Mészáros, A. Varga, and M. Kirsche, "INET framework," in *Recent Advances in Network Simulation*. Springer, 2019, pp. 55–106.
[45] A. Varga, "OMNeT++," in *Modeling and Tools for Network Simulation*, K. Wehrle, M. Güneş, and J. Gross, Eds. Heidelberg, Germany: Springer, 2010, pp. 35–59.
[46] H.-J. Kim, M.-H. Choi, M.-H. Kim, and S. Lee, "Development of an Ethernet-based heuristic time-sensitive networking scheduling algorithm for real-time in-vehicle data transmission," *Electronics*, vol. 10, no. 2, p. 157, Jan. 2021.
[47] L. Leonardi, L. L. Bello, and G. Patti, "Performance assessment of the IEEE 802.1Qch in an automotive scenario," in *Proc. AEIT Int. Conf. Electr. Electron. Technol. for Automot. (AEIT AUTOMOTIVE)*, Nov. 2020, pp. 1–6.
[48] *IEEE Standard for Local and Metropolitan Area Networks—Frame Replication and Elimination for Reliability*, IEEE Standard 802.1CB-2017, 2017, pp. 1–102.
[49] O. Alparslan, S. Arakawa, and M. Murata, "Next generation intra-vehicle backbone network architectures," in *Proc. IEEE 22nd Int. Conf. High Perform. Switching Routing (HPSR)*, Jun. 2021, pp. 1–7.
[50] S. Ettinger, S. Cheng, B. Caine, C. Liu, H. Zhao, S. Pradhan, Y. Chai, B. Sapp, C. R. Qi, Y. Zhou, Z. Yang, A. Chouard, P. Sun, J. Ngiam, V. Vasudevan, A. McCauley, J. Shlens, and D. Anguelov, "Large scale interactive motion forecasting for autonomous driving: The Waymo open motion dataset," in *Proc. IEEE/CVF Int. Conf. Comput. Vis. (ICCV)*, Oct. 2021, pp. 9710–9719.
[51] W. Maddern, G. Pascoe, C. Linegar, and P. Newman, "1 Year, 1000 km: The Oxford robotcar dataset," *Int. J. Robot. Res.*, vol. 36, no. 1, pp. 3–15, 2017, doi: 10.1177/0278364916679498.
[52] *Milan Whitepaper*. Accessed: Jul. 14, 2022. [Online]. Available: https://avnu.org/wp-content/uploads/2014/05/Milan-Whitepaper_FINAL-1.pdf
[53] P. Emmerich, S. Gallenmüller, D. Raumer, F. Wohlfart, and G. Carle, "MoonGen: A scriptable high-speed packet generator," in *Proc. Internet Meas. Conf. (IMC)*, Tokyo, Japan, Oct. 2015, pp. 275–287.
[54] L. Torvalds. *Linux Kernel V5.0*. Accessed: May 19, 2022. [Online]. Available: https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/?id=v5.0
[55] R. Love, "Kernel korner: CPU affinity," *Linux J.*, vol. 2003, no. 111, p. 8, 2003.
[56] R. Delgado and B. W. Choi, "New insights into the real-time performance of a multicore processor," *IEEE Access*, vol. 8, pp. 186199–186211, 2020.
[57] A. C. Heursch, D. Grambow, A. Horstkotte, and H. Rzehak, "Steps towards a fully preemptable Linux kernel," *Memory*, vol. 48, p. 31, 2003.
[58] D. T. Marr, F. Binns, D. L. Hill, G. Hinton, D. A. Koufaty, J. A. Miller, and M. Upton, "Hyper-threading technology architecture and microarchitecture," *Intel Technol. J.*, vol. 6, no. 1, pp. 1–12, 2002.
[59] J. Charles, P. Jassi, N. S. Ananth, A. Sadat, and A. Fedorova, "Evaluation of the Intel CoreT i7 turbo boost feature," in *Proc. IEEE Int. Symp. Workload Characterization (IISWC)*, Oct. 2009, pp. 188–197.
[60] Linux Foundation. *Cyclictest*. Accessed: Jul. 14, 2022. [Online]. Available: https://wiki.linuxfoundation.org/realtime/documentation/howto/tools/cyclictest/
[61] D. M. Ingram, P. Schaub, D. A. Campbell, and R. R. Taylor, "Performance analysis of PTP components for IEC 61850 process bus applications," *IEEE Trans. Instrum. Meas.*, vol. 62, no. 4, pp. 710–719, Apr. 2013.
[62] *I210_Datasheet_V_3_7.pdf*. Accessed: Jun. 30, 2022. [Online]. Available: https://www.mouser.com/datasheet/2/612/i210_ethernet_controller_datasheet-257785.pdf

MARCIN BOSK received the bachelor's and master's degrees in computer engineering from the Technical University of Berlin, in 2018 and 2020, respectively. He is currently pursuing the Ph.D. degree with the Chair of Connected Mobility, Technical University of Munich. He is also a Researcher with the Chair of Connected Mobility, Technical University of Munich. His research interests include time-sensitive and in-vehicular networks, with special interest on novel and cross-layer approaches to their realization. He is also interested in the 5G and beyond mobile network architecture, especially considering network slicing and QoE-aware network management.

FILIP REZABEK received the Master of Science degree in communications engineering from the Technical University of Munich, in 2020. He is currently pursuing the Ph.D. degree with the Chair of Network Architectures and Services. He is currently the Chair of Network Architectures and Services, Technical University of Munich. His research interests include network security, applied and threshold cryptography, and distributed systems resilience and robustness. Besides, he is active in the area of TSN with focus on intra-vehicular networks and smart manufacturing. For both areas are important aspects of reproducible experiments.

KILIAN HOLZINGER is currently pursuing the Ph.D. degree with the Chair of Network Architectures and Services, Technical University of Munich (TUM). He is also a Research Associate at the Chair of Network Architectures and Services, TUM. He studied at TUM and the University of the French Antilles. His research interests include time deterministic, reliable, and resilient network architectures with a special focus on monitoring and telemetry.

ANGELA GONZALEZ MARINO received the bachelor's degree in telecommunications engineering from the Universidade de Vigo (UVIGO), Vigo, Spain, in 2015, and the master's degree in electronics engineering systems from the Universidad Politecnica de Madrid (UPM), Madrid, Spain, in 2016. She is currently pursuing the Ph.D. degree with the Universitat Politècnica de Catalunya (UPC), Barcelona, Spain. She was worked at HP Inc., Barcelona, as a Research and Development Electronics Engineer, from 2016 to 2020, designing electronics for large format printers and supporting the full product lifecycle development. She is currently with the Automotive Engineering Laboratory, Huawei Technologies, Munich Research Center, Munich, Germany, focusing on HW accelerators design for automotive networking solutions. Her current research interests include HW design for automotive in-vehicle networks and system on chip design.

ABDOUL AZIZ KANE received the master's degree in electrical engineering with a specialization/focus on embedded electronics from the ESCPE Lyon, France, in 2013. He currently works at the Huawei Technology's Munich Research Center as a Principal Functional Safety Researcher. Previously, he worked at Infineon Technologies as an Application and Concept Engineer for automotive microcontrollers. His research interests include the conceptualization and design of safe in-car communication networks for automated driving.

FRANCESC FONS (Senior Member, IEEE) received the bachelor's degree in electrical engineering, the master's degree in automatic control and industrial electronics engineering, and the Ph.D. degree in electronics technology from Universitat Rovira i Virgili (URV), Tarragona, Spain, in 1995, 2001, and 2012, respectively. He has focused his professional career on the automotive electronics industry, working on research and development in the areas of embedded software, systems, hardware, and networks. Along his career, he has been with different automotive Tier 1 and Tier 2 suppliers from USA, Germany, and China, and has participated in the successful launch of many commercial products for OEMs in Europe and Asia. He is currently with the Automotive Engineering Laboratory, Huawei Technologies, Munich Research Center.

JÖRG OTT received the Diploma degree in computer science from TU Berlin, in 1991, the Diploma degree in industrial engineering from TFH Berlin, in 1995, and the Ph.D. degree from TU Berlin, in 1997. He has been the Chair of Connected Mobility at the Faculty of Informatics, Technische Universität München, since August 2015. He was a Full Professor in networking technology at Aalto University, from 2005 to 2015, and an Adjunct Professor, from 2015 to 2021. His research interests include network architectures, (real-time) (transport) protocols, and algorithms for connecting mobile nodes to the internet and to each other. He explores edge and in-network computing and decentralized services for achieving robustness and privacy.

GEORG CARLE received the degree from the University of Stuttgart, the degree from Brunel University, London, the degree from the Ecole Nationale Superieure des Telecommunications, Paris, and the Ph.D. degree in computer science from the University of Karlsruhe, in 1996. He has been a Professor at the Department of Informatics, Technical University of Munich, since 2008. He is currently holding the Chair of Network Architectures and Services. He worked as a Postdoctoral Scientist at the Institut Eurecom, Sophia Antipolis, France, and Fraunhofer Institute for Open Communication Systems, Berlin. From 2003 to 2008, he was a Professor at the University of Tubingen.