Journal of Network and Systems Management (2022) 30: 74
https://doi.org/10.1007/s10922-022-09686-0

EnGINE: Flexible Research Infrastructure for Reliable and Scalable Time Sensitive Networks

Filip Rezabek[1] · Marcin Bosk[1] · Thomas Paul[1] · Kilian Holzinger[1] · Sebastian Gallenmüller[1] · Angela Gonzalez[2] · Abdoul Kane[2] · Francesc Fons[2] · Zhang Haigang[2] · Georg Carle[1] · Jörg Ott[1]

Received: 18 February 2022 / Accepted: 8 August 2022 / Published online: 8 September 2022

Abstract

Self-driving and multimedia systems have common implications: increased demand on network bandwidth and computation nodes. To cope with the current and future challenges, intra-vehicular networks (IVNs) change their layout. They are built around powerful central nodes connected to the rest of the vehicle via Ethernet. The usage of Ethernet presents a challenge, as it by design lacks support for deterministic behavior, which is crucial for real-time systems. Therefore, the IEEE Time-Sensitive Networking (TSN) task group offers standards introducing low-latency and deterministic communication into Ethernet based networks allowing coexistence of best-effort and real-time traffic. To understand the coexistence challenges, these new networked systems need to be thoroughly evaluated with IVN requirements in mind. To assess various topologies, configurations, and data traffic types in IVN setups, we introduce *En*vironment for *G*eneric *I*n-vehicular *N*etworking *E*xperiments—*EnGINE*. It allows, among many others, repeatable, reproducible, and replicable TSN experiments with high precision and flexibility. *EnGINE* is based on commercial off-the-shelf hardware and uses the flexible Ansible framework for experiment orchestration. This allows us to configure various topologies emulating realistic behavior of IVNs or other time sensitive systems used, e.g., in industrial automation. Obtaining such realism is challenging using simulations. Based on available related work, we further address the challenges found in those networks, especially IVNs. We derive TSN domain framework requirements, provide details on design decisions for the *EnGINE*, and present results to show its capabilities. The results present relevant network metrics based on collected data. A key focus is on the experiment campaigns realism achieved by real IVNs' data footage and the OS optimizations to offer real-time behavior. We believe that *EnGINE* provides the ideal environment for TSN experiments from different domains.

Keywords TSN · IVN · Reproducibility · Replicability · Experiments

Filip Rezabek and Marcin Bosk are joint first authors.

Extended author information available on the last page of the article

1 Introduction

Autonomous driving, new connectivity services, over-the-air upgrades, shared mobility: These are just a few recent trends in the automotive industry. A common factor enabling these technologies is a secure, fast, and reliable intra-vehicular network (IVN). Indeed, we are now seeing more and more Ethernet-based solutions aiming to fulfill these requirements. Although, by design, Ethernet does not offer deterministic behavior, the Time-Sensitive Networking (TSN) family of standards provides real-time guarantees to Ethernet.[1]

Performance and capabilities of TSN have been a research subject in recent years, mostly conducted in simulation environments. Simulations have multiple advantages, such as a fast development cycle, ease to reproduce and configure, and high flexibility. However, they often show far from realistic traffic behavior as real deployments artifacts are omitted, e.g., clock deviation. This poses a challenge, as real-world deployments present such artifacts and it is important to account for them during the system design. Therefore, we introduce a solution that combines simulations' advantages while deployed on a physical topology containing machines emulating zonal gateways (ZGWs) [1] and vehicle control computers (VCCs) [2]. The ZGWs are used as interconnections among different subnetworks and technologies within the IVN, whereas VCCs can be considered as highly connected, higher performance computation nodes. This solution aims to evaluate new generations of IVNs. We want to study the impact of growing data volume on application and network performance, and determine suitable network component distribution and interconnections in an automated manner. Besides, we want to keep visibility on corner cases.

This paper builds on top of already published results introducing the *EnGINE* framework (**En**vironment for **G**eneric **I**n-vehicular **N**etwork **E**xperiments) and extends it by new insights [3]. Namely, we provide new insights regarding integration and configuration of network setup with respect to Linux networking stack and traffic forwarding, overview of used tools, optimizations of operating system (OS), and showcase a complex use-case deployed using the framework. The framework offers various configurations for queuing disciplines and TSN capable commercial off-the-shelf (COTS) network interface cards (NICs). In *EnGINE* we initially focus on IEEE 802.1Qav [4], IEEE 802.1Qbv [5], and IEEE 802.1AS [6] standards with potential for extension and also inclusion of higher-layer TSN capabilities.

To manage the infrastructure, we use an orchestration tool built on *Ansible*.[2] It brings flexibility to network and data sources configuration. Moreover, we monitor and record events for further evaluation or traffic re-play in the network to identify architecture limits. The experiments run without human interaction, can be reproduced, and are easily configured. As reliability is another essential characteristic of the automotive networks, with *EnGINE*, we can inject various malfunctions to test packet loss and link failures.

[1] https://www.ieee802.org/1/pages/tsn.html, Accessed 07 Jan 22.
[2] https://www.ansible.com, Accessed 25 Nov 21.

The assessed metrics and data traffic patterns follow the recommendations presented by the AVNU Alliance for the individual stream reservation (SR) classes. AVNU Alliance aims to create an ecosystem servicing the precise timing and low latency requirements for automotive and other diverse applications using open standards. It introduces SR classes and their prioritization [7]. The used TSN standards follow the recommendation of IEEE P802.1DG TSN Profile for Automotive In-Vehicle Ethernet Communications [8].

This paper presents our approach to building a configurable and flexible infrastructure fulfilling the unique IVN needs. We define requirements for an IVN testbed and describe the means to achieve them, including the toolchain to execute network experiments. Furthermore, we present our main tools and software (SW) for conducting various network experiments to achieve deterministic behavior using Ethernet. Lastly, we introduce use-cases that can be tested, show *EnGINE* sample configuration, and the exemplary results the framework makes.

The paper follows a standard structure where in Sect. 2 we provide information on related work in the domain and give an overview of relevant TSN standards. The section is followed by Sect. 3 in which we derive the requirements placed on the *EnGINE*. Defined requirements are taken into account for design, details of which we introduce in Sect. 4. To showcase the *EnGINE* capabilities, we present a sample use-case with configuration and results in Sect. 5. Since during the use of *EnGINE* we also identified a few limitations, we describe them in Sect. 6. Finally, we summarize and provide insights for future work in Sect. 7.

2 Background and Related Work

As introduced in [1, 9], future IVNs have to deal with larger transferred data volume due to focus on advanced driver-assistance systems (ADAS) and multi-media functions in the vehicles. An example of the throughput required for these systems is shown in [10]. Manufacturers cope with those challenges by shifting to Ethernet, which is inexpensive and well understood from classical IT and telecommunication systems. This brings an advantage during development, as classical applications can be easily ported to the intra-vehicle domain. Unfortunately, by design, Ethernet is not suitable for vehicles as IVNs have strong requirements for real-time performance and guarantees. Therefore, two prominent solutions, IEEE Audio-Video Bridging [11], now known as the TSN working group, and TTEthernet [12], are proposed introducing deterministic behavior to Ethernet.

In recent years, we have seen various activities in this domain focused on the evaluation of individual standards on commodity or proprietary hardware (HW) [13, 14], modeling of TSN standards [15–17], and simulations [18–21]. Unfortunately, many publications evaluating performance on physical devices rely on custom HW [13] or use simple setups containing only a few nodes [14]. On the other hand, simulations introduce setups with a large number of nodes and data flows, as

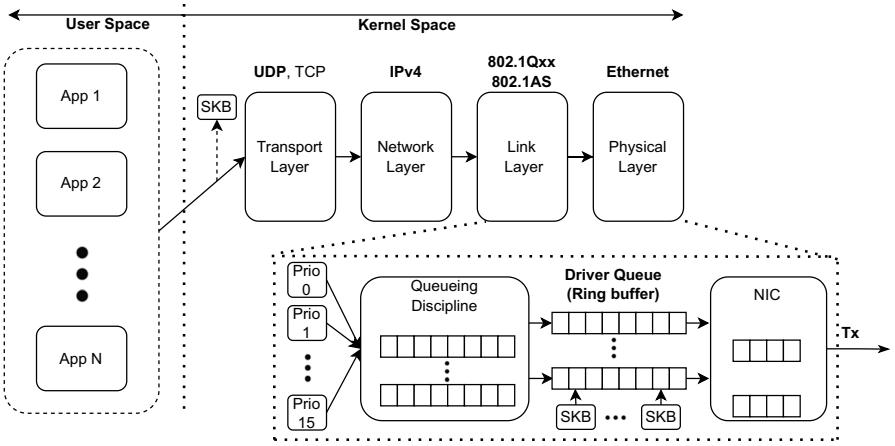


Fig. 1 Simplified overview of the Linux Networking Stack

shown in [21], which uses Real-Time at Work (RTaW) Pegase commercial solution[3]. Similarly, the open-source simulator OMNeT++ [22] is used in other works [18, 19] where two significant plugins offering TSN are considered, Core4Inet [23] and NeSTiNg [24].

Finally, we start to see a paradigm shift in IVNs where configuration and logic decisions are no longer handled on individual nodes but rely on a central controller leading towards Software-Defined Networks [25–28]. The central controller can be essential for real-time reconfiguration of the network, offering higher system reliability. To satisfy real-time guarantees, the system has to reconfigure and setup communication in less than 100 ms or even 50 ms, which might not be possible with traditional link-layer protocols [29, 30].

2.1 Networking Stack

The Linux networking stack used within the *EnGINE* framework is presented in Fig. 1, showing how individual traffic originating from user space applications reaches NIC's interface. The operation starts in user space, where an application creates data to be sent out on the network. This payload is passed to the kernel space using a system call. At that point, a socket buffer (SKB) storing the data and additional metadata of a given packet is created. This structure then passes through transport, network, and link layers, at which relevant headers are added for used protocols, such as UDP, IPv4, and MAC. Of note, the TSN standards relevant within the publication's scope are implemented on link-layer as part of the queuing disciplines (qdiscs). Qdiscs are implemented using parent-child hierarchy, where parent qdisc can have several child qdiscs configured under it. This allows to combine various types of qdiscs. Before a packet reaches the qdiscs, its priority is determined based

[3] https://www.realtimeatwork.com/rtaw-pegase/, Accessed 06 Jan 22).

on the information stored in the SKB. These priorities are mapped to individual traffic classes (TCs), which are mapped to interface queues on a given port. The qdisc determines when the packet is forwarded to the driver queue, also known as a ring buffer. Packets are then read by the NIC and stored in the HW queue before they are processed and dequeued on the wire.

Qdiscs and TSN Standards are used to achieve the desired latency and jitter within *EnGINE*. Linux implements several synchronous TSN standards as a part of the qdiscs. We focus on IEEE 802.1Qav and IEEE 802.1Qbv standards as considered in the IEEE P802.1DG TSN Profile for Automotive In-Vehicle Ethernet Communications [8]. IEEE 802.1Qbv is enabled by the Precision Time Protocol (PTP). In the following, we give an overview of these standards and introduce their basic functionality.

IEEE 802.1Qbv [5] Traffic Scheduling. IEEE 802.1Qbv is also known as Time-Aware Shaper (TAS), or Time Aware Priority Shaper (TAPRIO) qdisc in Linux, and "Enhancements for scheduled traffic" as a part of the IEEE 802.1Q-2018 [31] standard. It provides support for synchronized scheduling of multiple TCs on a single interface. The traffic flows are controlled by gates that operate according to a cycle determined by the system configuration. This allows various TCs to have a dedicated transmission window within a cycle of configurable length. During this window, the gate of the given class is open. Packets can be sent only when a corresponding gate is open and enough time for the transmission remains until gate closes. This operation results in TCs that are fully separated in the time-domain and have pre-defined transmission window opportunities.

In Linux TAPRIO qdisc, the TCs are mapped to HW queues and their respective transmission windows using their assigned priorities. TAPRIO is enabled by the Earliest TxTime First (ETF) qdisc[4] configured for each HW queue of the NIC. ETF enables the application to control the transmission time of its packets. Therefore, it is sometimes also referred to as a "LaunchTime" feature. The packets are sorted by their pre-defined transmission time within each queue and are kept in the queue by the qdisc until this deadline arrives. Some NICs, including the Intel I210 used extensively in *EnGINE*, also support the "LaunchTime" feature, which, when enabled, allows the frame to be transmitted, controlled by the NICs HW, at the specified time.

IEEE 802.1Qav [4] Traffic Scheduling. It is a a part of the IEEE 802.1Q-2018 [31] standard known as the "Credit-based shaper" (CBS) algorithm that is used to allocate bandwidth to SR classes. The algorithm protects the allocation for each SR class using a scheduling system based on credits, where start of a transmission is allowed only when the collected credit is ≥ 0. A class accumulates tokens at a rate specified by the *idleSlope* parameter, while frames are being queued in the corresponding queue. When there are no more frames waiting in a SR class queue and its credit was > 0, its available credit is set to 0.

Within Linux, CBS is usually combined with Multiqueue Priority Qdisc (MQPRIO).[5] The discipline enables mapping of SR classes, defined by their assigned priorities, into

[4] http://man7.org/linux/man-pages/man8/tc-etf.8.html, Accessed 22 Jan 22.

[5] http://man7.org/linux/man-pages/man8/tc-mqprio.8.html, Accessed 22 Jan 22.

HW queues of the NIC. The pre-defined classes are then associated with CBS configured per HW queue as MQPRIO's child qdiscs.

IEEE 1588 [32] standard introduces PTP for precise time synchronization in any networked system. Clocks are synchronized via PTP instances which are running on each participating device. The devices are organized in a master-slave hierarchy. A slave synchronizes its clock with a master by exchanging messages over the network. At the top of this hierarchy sits a grandmaster (GM) clock, which determines the reference time for the whole system.

There are five types of PTP devices: Ordinary Clock, Boundary Clock, End-to-end Transparent Clock, Peer-to-peer Transparent Clock, and PTP Management Node. The ordinary clock contains only one PTP port and thus can only be a grandmaster or a slave PTP instance. The Boundary clock contains multiple PTP ports each behaving as ports in the ordinary clock. It can become the grandmaster, but does forward any PTP messages. In contrast to the Ordinary and Boundary clocks, the End-to-end and Peer-to-peer Transparent Clocks do not synchronize the clock of the machine they operate on. These devices only measure the correction that needs to be applied to each PTP packet and forward all PTP packets. Ordinary and transparent clocks may be combined. The PTP Management Node serves as a human management interface, has several PTP connections and may be combined with any other PTP device type.

IEEE 802.1AS [6] standard uses methods defined in IEEE 1588 and applies these to the concept of Time-Sensitive Networking in the form of a generic Precision Time Protocol (gPTP). Its main difference to PTP is that the messages are only exchanged 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 ordinary clock, whereas the relay instance is equivalent to a transparent clock. In gPTP, the packets are exchanged only between PTP instances, that is non-PTP devices cannot be used to forward PTP packets. Furthermore, all devices in gPTP must use a clock with the same frequency.

All standards mentioned above are considered in the IEEE P802.1DG TSN Profile for Automotive In-Vehicle Ethernet Communications [8].

3 Analysis

Independent reproduction and verification of results is non-trivial. Even though ACM Policy considers reproducibility as a three-stage process [33], its adoption is still in the early stages. Reproducible research in the domain of computer networking has been a continuous activity [34–37]. Thus, we decide to continue this approach when building *EnGINE*. As currently there is no easy way to verify results within the scope of IVN, we define *EnGINE* with a focus on IVN. Nevertheless, the functionality can easily be extended for any real-time sensitive domains.

To realize this approach and achieve desired flexibility of *EnGINE*, we identify a set of requirements **R**, which the framework should fulfill in order to handle various experiments relevant in the IVN, and by extension also TSN domains:

R_1 Repeatability – experiments can be easily repeated using the same setup within the same organization [33]
R_2 Reproducibility – experiments can be easily reproduced by original organization and external parties using the same setup [33]
R_3 Replicability – experiments can be easily reproduced by original organization and external parties using different setups with same capabilities [33]
R_4 Configurability – choice of experiments and their parameters can be easily configured and deployed
R_5 Autonomy – experiments run without human interaction
R_6 Interpretability – generated artifacts can be analyzed and explained
R_7 Realism – works with real-world traffic patterns and provides results comparable to real-world deployments
R_8 Scalability – the network can handle large amount of traffic and various number of nodes
R_9 Reliability – the system can handle HW/SW malfunctions
R_{10} Diversity – the framework can handle a variety of input data formats, deployment scenarios, and areas of application
R_{11} Affordability/Accessibility – the framework does not rely on proprietary solutions which might be less accessible to other organizations
R_{12} Openness – the framework and its underlying infrastructure are built using open-source and easily accessible solutions
R_{13} Updateability/Upgradeability – the components of the infrastructure can be easily updated or upgraded to satisfy new requirements

Requirements $\mathbf{R}_1$–$\mathbf{R}_3$ cover the focus on TSN and IVN infrastructure. The aim is to provide an environment that is transparent and it, as well as the conducted experiments, can be reproduced by all parties [33]. $\mathbf{R}_4$–$\mathbf{R}_6$ are relevant from the usability and experiment preparation perspectives. These cover experiment configuration, description, and autonomous execution, as well as the interpretation of the collected artifacts, such as packet captures or logs.

Based on the overview of IVNs, we derive requirements $\mathbf{R}_7$–$\mathbf{R}_{10}$. $\mathbf{R}_7$ focuses on realistic representation of data traffic patterns present in IVNs and other real-time applications due to the large scale of available data sources. Traffic patterns directly affect the network performance and are crucial for the proper configuration of TSN. In [10], we can see an overview of such traffic streams and various data sources which motivates $\mathbf{R}_{10}$. Besides, in IVNs or other time critical use-cases we see the usage of real-time OS offering deterministic behavior to applications execution. Therefore, we are interested in ways to offer real-time behavior with Linux. Furthermore, considering $\mathbf{R}_{10}$, the framework should not only be limited to IVNs but also be extensible towards industrial automation as well as other use-cases with real-time requirements. Similarly, $\mathbf{R}_8$ aims to cover the use-cases based on the type of vehicle and manufacturer. The IVN joins several ZGWs, gateway controllers, and VCCs that are interconnected in various topologies. Also, the number of sensors varies, resulting in a wide range of data traffic volumes. The complexity of deployments also applies to other real-time environments. Besides, with the management host is the overhead on individual nodes during the experiment is minimal which contributes

to its scalability. With $\mathbf{R}_9$, the framework shall offer capabilities to emulate malfunctions on various levels in the infrastructure in order to deliver low failure rates.

$\mathbf{R}_{11}$–$\mathbf{R}_{13}$ address the fact, that a lot of research is done on proprietary solutions. That makes it challenging for other teams to work with and achieve ACM policy recommendations [33]. Similarly, proprietary solutions make upgrades to the latest technologies financially demanding.

We also identified additional requirements, which were not selected as the primary focus. Current IVNs and industrial automation systems are heterogeneous as they contain various network technologies, such as CAN, LIN, MOST, FlexRay, and Ethernet [1, 9]. We did not consider other technologies and focus purely on Ethernet, which is a backbone of modern IVN and TSN environments. Other solutions might be present in other parts of the network. These solutions may be interconnected via a gateway, which can translate to Ethernet [26].

Focusing on IVNs, we analyze available HW and SW components, management tools, and network control mechanisms considering the defined requirements. Details on the developed infrastructure and its design are provided in the following section.

4 Design

The primary goal of *EnGINE* is to provide an all-in-one solution for reproducible IVN experiments. Based on the analysis performed in Sect. 3, the final implementation of the architecture has to fulfill the set of requirements **R**.

As shown in Fig. 2, an experiment within the *EnGINE* framework consists of three elements: the **input**, which defines the traffic type and scenario under which the network is tested; the **System Under Test** (SUT), including all networked infrastructure used in an experiment. The network structure can be configured for different topologies using various network protocols and scheduling strategies. Finally, the experiments result in an **output** which may be physical actuation or creation of artifacts recorded within the SUT.

4.1 Architecture

We base *EnGINE* on COTS HW. It comprises twelve ZGWs and three VCCs. The HW configuration of each type is shown in Table 1. Using this approach, we satisfy requirements $\mathbf{R}_{11}$ and $\mathbf{R}_{12}$ and to an extent also the $\mathbf{R}_3$, as other teams can replicate similar scenarios using easily accessible solutions.

We select Intel® I210, I350, and I225 NICs for their HW support of various TSN standards, as shown in Table 1. Nevertheless, even if a NIC does not support TSN standards we can use SW support of qdiscs. To cope with the always increasing throughput requirement, we also use Intel® X552, which is a 10GbE NIC and supports IEEE 802.1AS, but no additional TSN standards. The NIC can still be used to evaluate the impact of purely SW based TSN hop with remaining hops supporting at least some TSN standards in HW. For some experiments where high clock precision

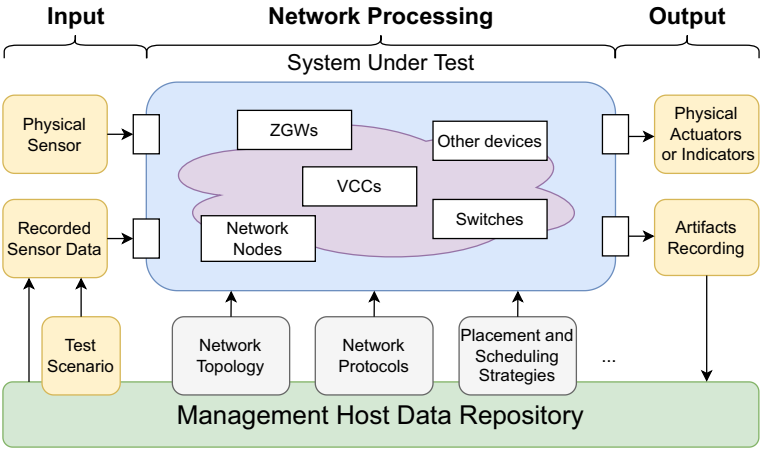


Fig. 2 Overview of experiment components

Table 1 Hardware used for VCCs and ZGWs with details of supported TSN standards by NICs

	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[a]	4×1 GbE Intel I210[a]
	4×1 GbE Intel I350[b]	1×2.5 GbE Intel I225[a]
	2×10 GbE Intel X552[b]	2×10 GbE Cisco Nexus GM[b]

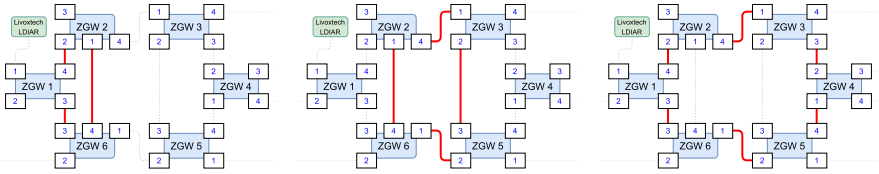
[a]IEEE 802.1Qav, Qbv, AS

[b]IEEE 802.1AS

is needed we are using Cisco Nexus GM SmartNIC supporting 10GbE and IEEE 802.1AS standard. The NIC, supports GPS clock synchronization and can serve as a GM in the network.

The VCCs and ZGWs are interconnected using the network adapters mentioned in Table 1. The network is structured in a way that allows for the testing of various in-vehicular system configurations. This enables the infrastructure to support various network complexities that can be found in different vehicle classes. As an example, we are able to configure networks using three, four, or six machines placed in a ring structure as presented in Fig. 3a–c respectively. With each node being equivalent to a ZGW, these correspond to networks found in low-, mid-, and high-end vehicles [1], fulfilling $\mathbf{R}_8$.

Shown configurations can be considered a part of the same vehicle platform as well. This configuration flexibility and additional availability of higher-bandwidth 2.5 Gbit/s and 10 Gbit/s connections satisfies requirement $\mathbf{R}_8$. Besides, the AVNU alliance recommends for stream reservation classes and their corresponding metrics various topologies containing up to 7 hops [7], which are also possible within the



(a) Low-End with 3 ZGW **(b)** Mid-End with 4 ZGW **(c)** High-End with 6 ZGW

Fig. 3 Sample configurations for various IVN complexities. Highlighted links used in respective configuration

infrastructure. To minimize the overhead on the individual nodes during the experiment run, the management hosts prepares the complete environment before the experiment starts.

Furthermore, *EnGINE* supports external traffic generators as well as sinks, currently demonstrated by the inclusion of a Livoxtech LIDAR Mid40 as a data source within the network topology [38]. Nevertheless, it is not practical even though it is possible to include more physical data sources. Therefore, we analyze options for using synthetic data that correspond to real-world traffic patterns that could be generated using traffic generators. Alternatively, packet captures can be generated from real-world sensor data that are then replayed in the network [10, 39]. Having both functionalities enables testing of real-world data sources transmitting data at scale over adjustable network topologies aiding in fulfillment of $\mathbf{R}_7$.

4.2 Configuration and Management

To manage and configure the infrastructure, we build a custom tool using Ansible, an open-source configuration management software. Ansible is an idempotent and descriptive language based on YAML and Jinja templates. It uses playbooks written in YAML files to express configurations and mapping of hosts to a set of roles. The management node runs individual playbooks, connects over Secure Shell (SSH) to experiment nodes, and executes individual playbook tasks. Figure 4 shows a typical communication, where ① the management host remotely executes commands on a node. Then ② the node runs this code and ③ interacts with other nodes. Afterwards, the nodes ④ store the collected artifacts on the management host, which ⑤ processes the collected artifacts.

Each experiment campaign is divided into four phases: **install**, **setup**, **scenario**, and **process** as shown in Fig. 5a. In the **install** phase, the nodes required for the campaign are allocated and booted with the OS of preference. For this step, we utilize the plain orchestration service (*pos*) [37].

Once the nodes are booted, the execution continues with **setup**. During this phase, the required prerequisites and packages are installed or copied from the management host. With all dependencies prepared, the nodes are ready to host individual experiment runs. In case a new version of a dependency package is released, the changes will be automatically applied on the test system to satisfy $\mathbf{R}_{13}$ as the dependencies are downloaded anew after each boot. To ensure better repeatability

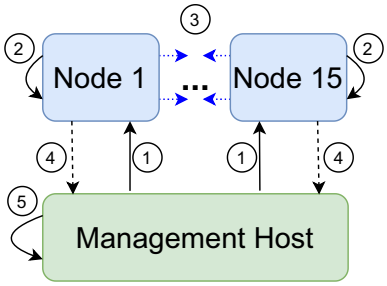
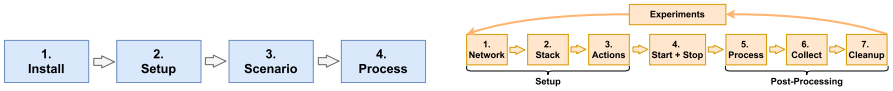


Fig. 4 Experiment workflow (cf. Gallenmüller et al. [34])



(a) Campaign phases **(b)** Individual experiment steps

Fig. 5 Experiment campaign overview

live images (run only in RAM) are used which ensure all residual states are erased once system is rebooted.

In the third phase, **scenario**, the individual experiments are conducted. Each experiment has to go through seven steps as described in Fig. 5b. First, the ***network*** topology is configured. The configuration is defined using a path between source and sink with individual hops and links along the way. Example topologies can be seen in Fig. 3. The paths are placed on the network using *Open vSwitch (OvS)* [40] and the priority of individual traffic is determined by the priority tag in the VLAN header. Each hop is set to forward the data towards its destination. Similarly, the ports can be configured with a traffic shaper of preference using Linux queuing disciplines configuration tool called *traffic control (tc)*. Currently the IEEE 802.1Qav and IEEE 802.1Qbv standards are supported. The framework could utilize other traffic shapers, e.g., Asynchronous Traffic Shaping, requiring their availability in Linux. To note, we have granular control over the configuration of the ports and can configure each individually. Besides, each NIC port has own physical HW clock, which is reflected in the PTP configuration using *linuxptp*.

Next, so-called ***stacks*** are instantiated on each node. A ***stack*** defines applications used during the individual experiment, such as traffic generator, packet captures, and others. To introduce dynamic behavior to the experiment, in step three, we may define additional ***actions***. These can include, for example, switching off a link or introducing an additional traffic path and thus satisfy requirement $\mathbf{R}_9$. Besides, this opens opportunities for a new set of use-cases focusing on evaluating the transition period between different states, such as network reconfiguration, application refresh, and others. To mention, the values we collect are limited only to the Linux-based OS, but the model describing the events can be used on different systems.

Finally, the experiment is ***started*** and then ***stopped*** after a configured timeout. During execution, there is no further interaction with the management host. The only

overhead that could affect the performance are regular status checks if the applications finished. Of note, these checks are performed outside of the isolated resources used for the experiment and do not impact the results in any way. After each experiment, the generated artifacts are ***processed*** on the individual node, ***collected*** and uploaded to the management host, and then ***cleaned up*** from the network nodes. To decrease the experiment execution time, the nodes do not have to be restarted and go through **install** and **setup** before the following experiment is executed.

Finally, after all experiments are successfully finished, the post-**processing** phase starts. Post-processing can be done either on individual or several experiments at once to collect different insights and understand the results better.

With the described approach, we are able to fulfill all goals set for *EnGINE*. We enforce a fixed configuration structure, which enables easy repetition of scenarios, thus satisfy requirements $\mathbf{R}_1$ and $\mathbf{R}_2$. Similarly, we have broad options of configuration of various network topologies and TSN parameters, applications stacks, and actions to evaluate the TSN behavior satisfying requirement $\mathbf{R}_4$. Besides, the individual scenarios, once properly configured, can run fully autonomously and at the end generate figures which provide insights into the experiment results satisfying both requirements $\mathbf{R}_5$ and $\mathbf{R}_6$.

The remaining requirements focus on specifics of IVNs and traffic present in them. To satisfy requirements $\mathbf{R}_7$ and $\mathbf{R}_{10}$ we use freely available datasets used for autonomous driving applications [10, 39] or synthetic data corresponding to traffic patterns [29, 41]. With this approach, we can emulate various traffic patterns present in vehicular networks and correspond to data sources available in the market. The corresponding traffic patterns can also be generated using traffic generators such as *Iperf3*, *send_udp*, or *MoonGen* [42]. The precision of generated traffic would be limited by the CPU processing and the Linux scheduler. Therefore, we mitigate these limitations using CPU isolation and affinity to approximate the real-time OS behavior. The data generated can be stored in the form of a packet capture using *tcpdump* for evaluation or even future replay. Since the used NICs support the IEEE 802.1AS standard, we can achieve high accuracy and precision of packet timestamps using HW timestamping.

4.3 Software Stacks

The execution of various experiments requires the use of numerous, open-source, tools. The applications are chosen to support the experiments, the configuration of the test environment, the framework itself, result capture, and result evaluation. As long as the applications can run in Linux environment, the *EnGINE* is generic enough to integrate easily new tools, applications, and middleware solutions that could extend the framework capabilities. Especially interesting would be middleware solutions present in the automotive domain, e.g., AUTOSAR. In the following, we introduce the most notable tools already integrated to the framework.

Generation of Data in *EnGINE* relies on various traffic generators to support a wide range of experiments. A specific generator is chosen depending on a given scenario and its specific requirements.

Iperf3[6] is a network performance measurement tool that is generally used to assess the network throughput. By default, it sends as many packets as possible between a server and a client instance to fully saturate the link. This application is used in the infrastructure to create a source and sink for network traffic. The packet size and a fixed throughput rate can be configured by command line parameters to generate different network patterns. The output log is used to detect any network limitations, e.g., throughput boundaries defined by TSN.

send_udp is a custom application inspired by the *udp_tai.c* application in TxTools.[7] This custom application allows low-level packet manipulation. It uses the socket APIs' *sendmsg*, *recvmsg*, and several socket flags, for instance, *SOF_TIMESTAMPING_TX_HARDWARE*, wrapped in C code to send and receive custom packets. Additionally, the packet metadata can be accessed and modified, thus, directly setting the packet priority in the send application and also defining a send timestamp in the future when the packet should be put on the wire by the NIC. The process of creating custom packets has a performance drawback. It is not possible to create as fast and as many packets as with *Iperf3*. However, with this application it is possible to utilize the Intel® LaunchTime feature present in Linux Kernel as ETF.

MoonGen is a high-performance, open-source SW packet generator based on the high-speed packet processing framework DPDK[8]. Using a single core with packets generated by user-defined Lua scripts, MoonGen can generate minimum sized packet at 10 Gbit/s (14.88 Mpps). MoonGen supports HW-assisted timestamping of packets and achieves precision in the order of sub-microseconds. This requires HW support by NICs using registers and timestamps required for the precision time protocol (PTP) [42].

Naturally, *EnGINE* is not limited to the above-mentioned traffic generators. Any traffic generation application can be considered and admitted to the network as a stack. The framework is also capable of accepting traffic from external data traffic sources, e.g., LIDAR [38].

TSN Configuration and Networking Tools are key parts of the *EnGINE* used for evaluation of various topologies and network configuration, so we introduce the most important ones.

tc[9] is a built-in Linux traffic control application we use to modify packet processing and queuing on the OS level. By changing the queueing disciplines and configuring various parameters we are able to control the packet transmission operation. On this level the time-sensitive networking behavior is applied to individual packet streams based on the priority value of the packets.

OvS is used on the infrastructure to separate the physical and logical network in order to test various topologies without manual intervention [40]. We use *OvS* to define flows as a way to individually route traffic over selected network nodes. An alternative method would have been to configure custom routes with the built-in

[6] https://iperf.fr/, Accessed 28 Jan 22.
[7] https://gist.github.com/jeez/bd3afeff081ba64a695008dd8215866f, Accessed 28 Jan 22.
[8] https://dpdk.org, Accessed 17 Jan 22.
[9] https://man7.org/linux/man-pages/man8/tc.8.html, Accessed 28 Jan 22.

Linux tools on every node of the network. We decided that beyond a certain number of traffic flows this approach becomes unfeasible and hard to manage. The *OvS* application is also collecting internal statistics about the packets passing through it which is another probe to extract data and get insights into the network state.

linuxptp[10] implements PTP according to the IEEE 1588 standard. It is one of the most prominent implementations for high precision time synchronization on Linux. *linuxptp* provides three tools. The *ptp4l* which is a daemon responsible for synchronization of the NIC HW clock within the gPTP domain. The *phc2sys*, being a daemon which synchronizes any two clocks within a system. Finally, the *pmc*, which is a utility enabling run-time configuration of *ptp4l* daemon. The framework uses *linuxptp* to synchronize HW clocks of the NICs.

cgroups[11] are a feature in Linux which allows for grouping of processes and control of resources each grouping can use. They are built into the kernel and can be controlled via "cgroupfs", a pseudo-filesystem. *cgroups* may also be organized in a hierarchical manner. In this work, we utilize this feature to assign flows of various applications into desired priorities. These priorities are used by the networking stack to determine which queue (and queuing discipline) the packets of a flow should be directed towards.

Data Capture & Visualization Tools are introduced next.

tcpdump[12] application is a well-known packet sniffer used to capture network packets. In the framework, it is used to collect incoming and outgoing network traffic and store it in a packet capture used for further analysis. The NICs available in the testbed support HW timestamping increasing the accuracy of the collected packet timestamps.

scapy[13] and *PyPacker*[14] (Python libraries) are used in a custom application to analyze the packets captured with *tcpdump*. Both *scapy* and *PyPacker* read each packet from the input file and extract certain information from the payload. *PyPacker* offers faster processing which is crucial for post-processing of large packet captures. Both tools are used for initial processing of packet captures before they are visualized for example with *GnuPlot* or *matplotlib*. As *scapy* is also capable of generating packet traces, it is a candidate for use as a traffic generator in the future extensions of the framework.

4.4 EnGINE Networking Stack

In Fig. 6 we see how packets generated in user space (e.g., LIDAR/RADAR or *iperf3*/*send_udp*) are passed through a firewall defined by *cgroups* towards a virtual interface. First, packet priorities are stored in the SKB. Then, a port on which an application runs on the transport layer is chosen. Furthermore, the source IP address

[10] http://linuxptp.sourceforge.net/, Accessed 28 Jan 22.
[11] https://man7.org/linux/man-pages/man7/cgroups.7.html, Accessed 28 Jan 22.
[12] https://www.tcpdump.org/, Accessed 22 Dec 21.
[13] https://github.com/secdev/scapy, Accessed 28 Jan 22.
[14] https://gitlab.com/mike01/pypacker/, Accessed 28 Jan 22.

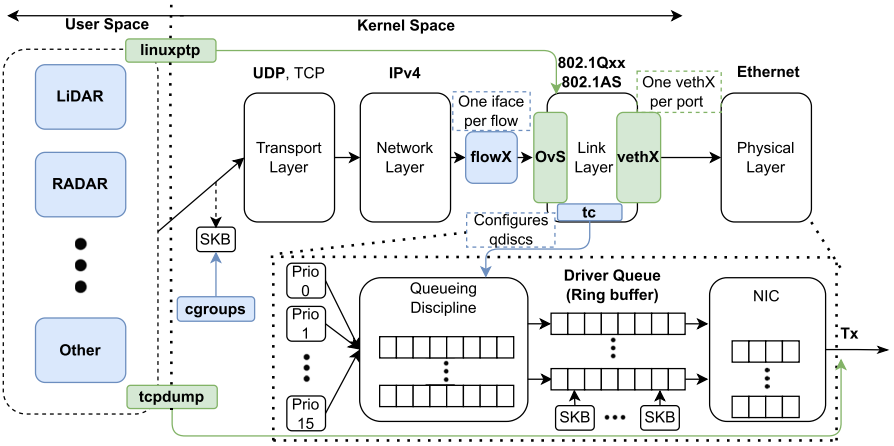


Fig. 6 Overview of EnGINE Application and Networking Stack

of a corresponding virtual interface *flowX* is used to create a socket <IP>:<PORT> pairing. Finally, each virtual *flowX* interface is connected to *OvS* where packets are forwarded to a single virtual interface (*vethX*) of a corresponding physical port.

The above information is used on the link layer to decide the flow path and the applicable TSN configuration. The source IP address specifies through which virtual interface *flowX* the application packets will go and therefore which flow path is followed. The packet's priority is determined by SKB priority information, which maps it to a stream and, therefore, TSN configuration. At the *vethX* VLAN headers are attached to the packets with the packet priority value being mapped to the VLAN Priority Code Point (PCP) header field before entering the Linux interface stack. This VLAN field determines with which priority the packet should be handled as it passes through the network and therefore which HW queue of the Intel® I210, I350, I225, or X552 NIC is used. Finally, *OvS* decides on which physical interface the packet is sent out.

To ensure clock synchronization, we use PTP implemented by *linuxptp*, which relies on *ptp4l* daemon that runs in the user space, but the clock synchronization messages exchanged among nodes in the system work on the link layer. To note, they could also rely on UDP with IPv4 or IPv6, but we rely on the link layer.

To forward the packets towards their destination using link-layer functionality, on each hop in the network we use *OvS*. Each node in the given experiment runs its own *OvS* session without using a central controller. Finally, packets can be captured on hops or the sink using *tcpdump* or, if interested in other metrics, we can use a suitable tool.

4.5 Operating System Optimizations

To achieve the best possible performance of *EnGINE*, we employ several techniques that allow us to minimize the impact of the OS on the experiments [36]. Those

optimizations include the use of a low-latency Ubuntu kernel, CPU isolation, CPU affinity, and using the CPU in "performance" energy management mode. With those settings, we aim to minimize any delay and jitter introduced by the OS, making the system better represent applications running on real ZGWs and VCCs, thus aiding in fulfillment of $\mathbf{R}_7$.

Low-latency Ubuntu Kernel adds several OS features allowing for reduction of the delay and jitter introduced by the process scheduling of the system [43]. Compared to a generic one, this kernel version considers interrupt requests (IRQ) from HW devices as threads. The threaded IRQs allow for manual configuration of the HW interrupt's priority. This setting is essential for the Intel® I210 and I225 NIC operation in *EnGINE* as it enables selection of real-time (RT) priority for IRQs coming from each of the four HW NIC queues. Another feature of the low-latency kernel is the introduction of preemption points, where the CPU scheduler actively looks for the higher priority threads that should be executed before other, longer tasks. The preemption also may involve interrupting execution of other threads to run higher priority tasks. In *EnGINE*, we use Ubuntu 20.04 LTS with GNU/Linux 5.4.0-90-lowlatency kernel. NIC IRQs are configured for RT priority. The priority configuration is done using *chrt*[15] tool which enables changes of thread properties.

CPU Affinity describes the ability to assign a thread to a specified CPU core [44]. There are two types of affinity, so called "soft" and "hard". The soft CPU affinity is the type usually employed by the CPU scheduler where it tries to keep the thread on the same core during run-time. There are no guarantees that the thread will not be moved to another CPU core. Hard affinity guarantees a thread's explicit binding to a CPU core, strictly respected by the CPU scheduler. In *EnGINE* we extensively use hard affinity to manually distribute threads and IRQs across the logical CPU cores that are necessary to keep low delay within the system. Considered functions we bind to specified cores include NIC IRQs and traffic generation applications such as *iperf3* or *send_udp*. Manual thread to CPU assignment is configured using the *taskset*[16] utility which enables CPU affinity management.

CPU Isolation of specified cores prevents the system scheduler from placing any user and OS tasks on the defined CPU core [45]. It allows us to manually prepare a number of cores where no other threads, except manually pre-defined ones using CPU affinity, are placed. In *EnGINE* this isolation prevents unwanted influence from other tasks that are not relevant to an experiment. The CPU cores are isolated using the *isolcpus*[17] system boot parameter.

CPU Configuration and Power Management options are further used to optimize the system for low-delay and low-jitter operation. We investigate settings that limit the CPU task execution overhead and prevent the system from going into low-power modes. Therefore, we include three configuration options that can be used to disable Simultaneous Multi-Threading (SMT), disable Automatic Overclocking, and

[15] https://man7.org/linux/man-pages/man1/chrt.1.html, Accessed 21 Jan 22.
[16] https://man7.org/linux/man-pages/man1/taskset.1.html, Accessed 21 Feb 22.
[17] https://www.linuxtopia.org/online_books/linux_kernel/kernel_configuration/re46.html, Accessed 21 Jan 22.

set the CPU Dynamic Voltage and Frequency Scaling (DVFS) governor to "performance" mode.

SMT [46] is a technology enabling simultaneous execution of two tasks on a single physical CPU core. This parallelism is achieved by duplicating the architectural state for the CPU core. The core appears as two separate ones for the OS despite sharing the same compute HW. SMT is aimed at improving the general performance of the system by more efficiently utilizing each physical CPU core. While the technology might have a negative impact on certain system and application performance, during testing we observed a decrease in performance with SMT disabled. Therefore, we use *EnGINE* with SMT enabled.

Automatic Overclocking, e.g., Intel's Turbo Boost [47], is a feature that dynamically increases the CPU clock frequency under high load in order to improve performance. The introduced variation of CPU clock frequency has the potential of increasing system jitter, however, in testing we have not observed such influence. Furthermore, the technology's sequential execution performance improvement positively affects the functionality of applications run during experiments. Thus, *EnGINE* usually operates with the feature enabled.

Linux allows selection of the CPU DVFS algorithms using *CPUFreq* governors [48]. The governors either set the CPU frequency to a pre-defined value, or allow the system to set the frequency dynamically according to current needs or load. Since we are using an Intel® processor, our governor selection is limited to two modes, namely "powersave" and "performance".[18] These modes either will or will not consider energy saving features of the CPU respectively. The power optimizations include aggressively lowering the CPU frequency when no load is present or putting the CPU into sleep states. In *EnGINE* we utilize the performance governor, limiting the changes in CPU frequency and thus minimizing OS induced jitter in our experiments.

5 Capabilities

EnGINE supports experiments ranging from small deployments of just two nodes and a single traffic flow, up to scenarios including thirteen nodes and numerous flows. Such flexibility and scalability allow us to evaluate various topologies corresponding to use-cases we can encounter in IVNs. In this section, we introduce a selected scenario to show and verify the configuration and capabilities of our framework. We demonstrate complete *EnGINE* functionality in a network of interconnected VCCs and ZGWs as shown in Fig. 7. The mid-end vehicle topology highlighted on Fig. 7 is used as a base for our use-case. The use-case we cover in detail comprises three flows, each with a path over a single, two, or three hops respectively. Depending on the stack configuration, different nodes act as a source or sink. To investigate this scenario, we define an experiment campaign using a combination of five individual YAML files (*00-nodes*, *01-network*, *02-stacks* , *03-actions*,

[18] https://www.kernel.org/doc/html/v4.19/admin-guide/pm/intel_pstate.html, Accessed 28 Feb 22.

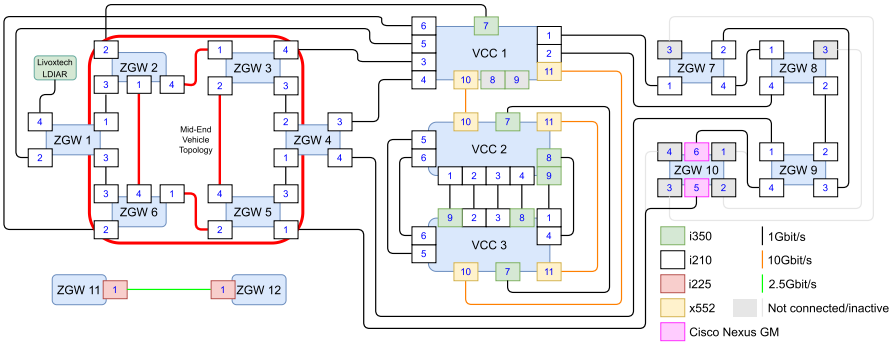
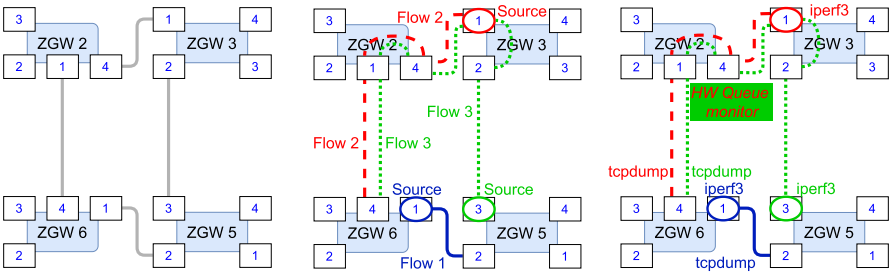


Fig. 7 EnGINE network overview



(a) Scenario nodes **(b)** Configured network **(c)** Configured stacks

Fig. 8 Overview of individual configuration steps impact on the experiment setup

and *04-experiments*). Impact of individual steps on the experiment campaign is presented in Fig. 8. The configuration is described in Sect. 5.1 and its outcome in Sect. 5.2.

Listing 1: 00-nodes.yml sample node mappings

```
---
use_low_latency_kernel: "yes" # Use low-latency kernel
num_isolated_cores: 4 # Isolate first 4 CPU cores
nodes:
  - zgw2 # Corresponds to Fig. 5
  - zgw3 # Corresponds to Fig. 5
  - zgw5 # Corresponds to Fig. 5
  - zgw6 # Corresponds to Fig. 5
node_mapping:
  node-1: zgw2
  node-2: zgw3
  node-3: zgw5
  node-4: zgw6
```

With *EnGINE* we can emulate real deployed IVN topologies. Such deployments can be identified from available related work and be ported to *EnGINE* for further investigation. The infrastructure can host more than seven hops between source and sink, supporting numerous flows, various traffic patterns generated live or replayed from recorded packet captures, and network topologies matching real-world scenarios. Such complex flow configuration would be reflected in the YAML files, as multiple combinations of traffic patterns and corresponding configurations of TAPRIO and CBS on the hops are needed. Moreover, we can focus on reliability, identifying how the system behaves in case of malfunctions, such as delay or link failures. Here, *03-actions.yml* plays a key role in introducing pre-defined malfunctions into the system.

Listing 2: 01-network.yml sample network configuration

```
---
network: # Configuration of networks for experiments
  net-1:
    tsn:
      tsn-1: ["node-1", "node-2", "node-3", "node-4"]
    flows:
      1: ":node-3:2,1:node-4:"
      2: ":node-2:1,4:node-1:1,4:node-4:"
      3: ":node-3:4,2:node-2:1,4:node-1:1,4:node-4:"
    num_net_cores: 3 # NIC IRQ deditcated cores
    nic_irq_rt: true # NIC IRQ real-time priority
  net-2:
    tsn:
      tsn-2: ["node-1:1", "node-2:1", "node-3:4", "node-4:1"]
    flows:
      1: ":node-3:2,1:node-4:"
      2: ":node-2:1,4:node-1:1,4:node-4:"
      3: ":node-3:4,2:node-2:1,4:node-1:1,4:node-4:"
    num_net_cores: 3 # NIC IRQ deditcated cores
    nic_irq_rt: true # NIC IRQ real-time priority
tsnconfigs: # TSN configurations for nodes/interfaces
  tsn-1:
    name: TAPRIO with ETF in Strict and Deadline mode
    taprio:
      txtime: true # Usage of txtime-assist mode
      delay: 400000 # txtime delay
      sched:
        - { queue: [1], duration: 300 }
        - { queue: [2], duration: 300 }
        - { queue: [3], duration: 400 }
    queues: # Configuration of individual HW queues
      1: { mode: etf, prio: [3], delta: 300000, deadline: yes, offload: no } #
          Supports HW offloading
      2: { mode: etf, prio: [2], delta: 300000, deadline: yes } # delta - fudge
           factor of ETF
      3: { mode: be, prio: [1, * ] }
  tsn-2:
    name: MQPRIO with CBS configured for 3 priorities
    taprio: {} # Empty since default MQPRIO used
    queues: # Configuration of individual HW queues
      1: { mode: cbs, prio: [3], idle:  100000 , send:  -900000 , high:  155 ,
           low:  -1125  }
      2: { mode: cbs, prio: [2], idle:  100000 , send:  -900000 , high:  297 ,
           low:  -1125  }
      4: { mode: be, prio: [1, * ] }
```

5.1 Configuration and Experiment Flow

To configure the selected use-case, we need to prepare the five YAML configuration files. During the preparation process we need to consider the desired outcome. In the following we describe the configuration files in more detail and reflect how they impact the experiment campaign phases. We start with *00-nodes.yml*, Listing 1, where we define the nodes and their mappings used during the whole experiment campaign. Here, we also specify parameters that determine how the nodes are booted. Among others, these parameters indicate whether a low latency kernel is used or how many CPU cores are isolated from the Linux scheduler, shown in lines 2 and 3 respectively.

Figure 8a highlights the subset of the used infrastructure onto which we map the use-case's experiment campaign configuration. The nodes defined in Listing 1 are used in the **install** phase to boot them with a predefined OS. The remaining nodes can be used for other experiment campaigns in parallel.

Before proceeding to **setup**, we need to specify network flows and TSN configuration in *01-network.yml*, Listing 2. We identified a configuration abstraction to have sufficient control over Linux ETF, TAPRIO, CBS, and MQPRIO qdiscs setup as well as individual HW queues of the NIC with corresponding traffic class priorities. Similarly, we can define on which nodes, or nodes' specific ports, a given TSN configuration is applied. Here, we also decide if and how many CPU cores are dedicated to NIC IRQs and whether those interrupts are set to use real-time priority.

The use-case configuration shown in Listing 2 consists of two parts, a network description starting in line 2 and a TSN configurations beginning on line 21. We configure two networks that encompass the same topology as in Fig. 8a. Each includes flows as outlined in Fig. 8b. The networks differ in their TSN configuration, i.e., `net-1` uses TAS and `net-2` uses CBS as queuing disciplines.

Of note is that we can configure TSN on the whole nodes as shown in line 5 or on individual interfaces of nodes as presented in line 14. The TSN configs `tsn-1` (line 22) and `tsn-2` (line 35) represent the configuration parameters of TAPRIO and CBS qdiscs respectively. For TAPRIO, we set a schedule specifying how long gates for each corresponding queue are open (lines 28-30). We configure the queues using ETF for queues 1 and 2 (lines 32, 33) and a best effort FIFO queue on queue 3 (line 34). ETF queues use priorities 3 and 2 respectively with the remaining priorities being assigned to the best effort queue. The fourth, remaining, HW queue is unused.

In `tsn-2`, similarly to `tsn-1`, we configure 3 queues (lines 39-41) for priorities 3 and 2 respectively using CBS set for 100 Mbit/s PHY layer throughput. The third queue (line 42) is configured for best effort traffic on all other priorities. Again, the fourth queue remains unused.

In the **setup** phase, PTP is also configured on all interfaces specified in the corresponding configuration file. After roughly 180 s (depending on HW performance), the nodes are ready to start with preparation of individual applications. This period is on purpose longer so the system stabilizes before starting with the experiment execution.

Next is the **scenario** phase where all experiments defined in a campaign are executed. We define the applications used in an experiment in *02-stacks.yml*, Listing 3.

Dependencies between applications, e.g., server-client, are incorporated by starting applications in sequence according to a specified level. The lower level indicates the earlier start of the application. The configuration parameters reflect the arguments with which the applications can be started. For the stacks in the presented use-case we are using three applications—`iperf`, `tcpdump`, and `queue_monitor`. Here, we can specify how much data is transmitted over the network. As an example for `iperf` (line 10) we specified packet payload `size` of 1180 B and `limit` of the traffic to 99.4 Mbit/s. This corresponds to 100 Mbit/s on the wire (including PHY layer headers) with packet spacing of 100 µs. Traffic volume also determines the run-time of the experiment or by the number of packets we want to send. We can select on which node individual application starts as shown in Fig. 8c, i.e., for `stack-2` this is shown on lines 17, 19, 22, and 27. Finally, to know where data shall be sent, we specify the flow number, which corresponds to the flow number in *01-network.yml* and is internally matched to a physical port of a respective node.

To collect data artifacts, we use in this case `tcpdump` and `queue_monitor`. For `tcpdump` the parameter `size` refers to how many bytes should be stored for each packet in the packet capture. Similarly, to limit the size of packet captures, we can select the `filter` option. For the `queue_monitor` we specify which queue types are of interest and on which interface they should be monitored.

Individual applications can terminate either after a timeout or upon completion. All applications must terminate gracefully to complete an experiment successfully. For completeness, *03-actions.yml* serves for definition of actions in the system, i.e., network interrupts but is not part of the given use-case.

The main logic is contained in *04-experiments.yml*, Listing 4. It references the information from the other YAML files to describe individual experiments. All entries are executed in sequential order. Before a new experiment starts, the old network configuration is flushed and previous processes killed to avoid disruptions of the next experiment run.

Listing 3: 02-stacks.yml sample stack configuration

```
---
stacks:
  stack-1:
    name:  Simple traffic scenario
    services:
      node-3:
        - { name: iperf, role: server, flow: 1, port: 1001, level: 0, signal:
            yes, use_core: 2 }
        - { name: tcpdump, flow: [1], size: 64, filter: "udp dst port 1001",
            file: "p1001", level: 0 }
      node-4:
        - { name: iperf, role: client, flow: 1, port: 1001, prio: 3, limit:
            94400000, size: 1180, level: 1, signal: yes, udp: yes, use_core: 2
             }
        - { name: tcpdump, flow: [1], size: 64, filter: "udp dst port 1001",
            file: "p1001", level: 0, use_core: 3 }
        - { name: queue_monitor, iface: [1], queue_types: "cbs,etf", level: 0,
            use_core: 3 }
    protocols: {}
  stack-2:
    name:  Complex traffic scenario
    services:
      node-1:
        - { name: queue_monitor, iface: [1], queue_types: "cbs,etf", level: 0 }
      node-2:
        - { name: iperf, role: client, flow: 2, port: 1002, prio: 2, limit:
            94400000, size: 1180, level: 1, signal: yes, udp: yes, use_core: 2
             }
        - { name: tcpdump, flow: [2], size: 64, filter: "udp dst port 1002",
            file: "p1002", level: 0 }
      node-3:
        - { name: iperf, role: server, flow: 1, port: 1001, level: 0, signal:
            yes, use_core: 3 }
        - { name: tcpdump, flow: [1], size: 64, filter: "udp dst port 1001",
            file: "p1001", level: 0 }
        - { name: iperf, role: client, flow: 3, port: 1003, prio: 3, limit:
            94400000, size: 1180, level: 1, signal: yes, udp: yes, use_core: 2
             }
        - { name: tcpdump, flow: [3], size: 64, filter: "udp dst port 1003",
            file: "p1003", level: 0 }
      node-4:
        - { name: iperf, role: client, flow: 1, port: 1001, prio: 3, limit:
            94400000, size: 1180, level: 1, signal: yes, udp: yes, use_core: 2
            }
        - { name: tcpdump, flow: [1], size: 64, filter: "udp dst port 1001",
            file: "p1001", level: 0 }
        - { name: iperf, role: server, flow: 2, port: 1002, level: 0, signal:
            yes, use_core: 3 }
        - { name: tcpdump, flow: [2], size: 64, filter: "udp dst port 1002",
            file: "p1002", level: 0 }
        - { name: iperf, role: server, flow: 3, port: 1003, level: 0, signal:
            yes, use_core: 3 }
        - { name: tcpdump, flow: [3], size: 64, filter: "udp dst port 1003",
            file: "p1003", level: 0 }
    protocols: {}
```

Listing 4: 04-experiment.yml sample experiment configuration

```
1 ---
2 experiments:
3   - { network: net-1 , stack: stack-1 , action: , signal: yes, timeout: 10,
        name: 1_simple-tas }
4   - { network: net-1 , stack: stack-2 , action: , signal: yes, timeout: 10,
        name: 2_complex-tas }
5   - { network: net-2 , stack: stack-1 , action: , signal: yes, timeout: 10,
        name: 3_simple-cbs }
6   - { network: net-2 , stack: stack-2 , action: , signal: yes, timeout: 10,
        name: 4_complex-cbs }
```

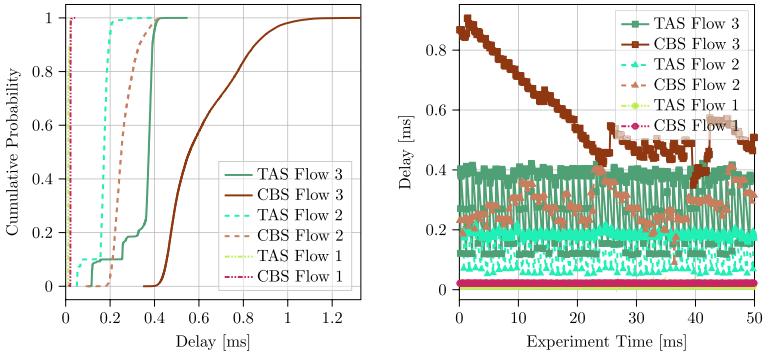
Listing 4 shows four configured experiments, the results of which are displayed in Sect. 5.2. We match each network with each stack to showcase two experiments focusing on *EnGINE* operation using the TAPRIO qdisc (Lines 3 and 4), and further two experiments focusing on the CBS qdisc (Lines 5 and 6).

After an experiment campaign is successfully completed, the **process** phase starts. The duration of this phase can take seconds in case of a low transferred traffic volume during an experiment or minutes/hours if large data sets need to be processed. We ensure the precision of collected data by using the NICs' HW timestamping capabilities.

5.2 Results

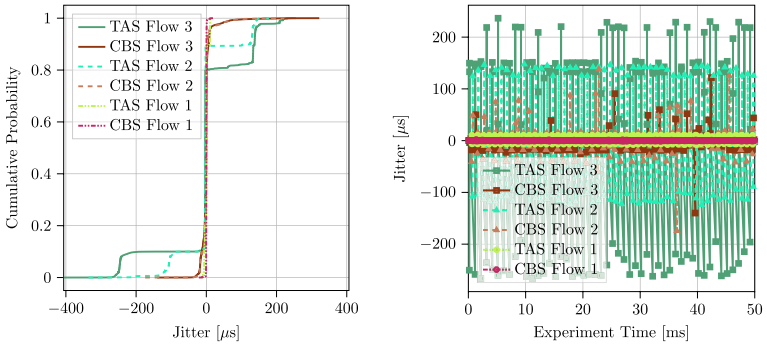
In this section, we showcase the evaluation capabilities of *EnGINE*. To show these features, we present the results of the use-case defined in Sect. 5.1. The presented figures constitute a subset of the evaluation materials that can be obtained using the framework. We include the results of the complex experiments defined in Listing 4 (lines 4 and 6) using `stack-2 - Complex traffic scenario`, Listing 3 (lines 14-34) and `net-1` for TAS and `net-2` CBS qdisc configuration. The generated traffic, topology, data flows, and collected artifacts are the same for both experiments. Finally, we present results of delay, jitter, and queue levels in the form of Empirical Cumulative Distribution Function (ECDF) and time series.

Delay is computed as $Delay_x = R_x - T_x$ where R_x is receive time, T_x sending time and x a packet index. These variables are available from packet capture on sender and receiver. We can use timestamps from two different machines to get an accurate value due to the clocks being synchronized with PTP. Packets on the receiver are timestamped using HW timestamp, which offers high precision. Besides, to correlate the packets on source and sink we can use the sequence number stored in the payload by *iperf3*. Jitter which is calculated as $Jitter_x = (R_x - R_{x-1}) - (T_x - T_{x-1})$ and indicates how much the end-to-end delay changed between two consecutive packets. Flow 1, 2, and 3 correspond to 1 hop, 2 hops, and 3 hops in the network respectively. The ECDF plots contain information from the approximately 100,000 packets collected over the 10 s of experiment duration. For better readability, the time series only shows delay for the first 50 ms of the experiment.



(a) ECDF **(b)** Time series

Fig. 9 End-to-end delay measured for selected use-case



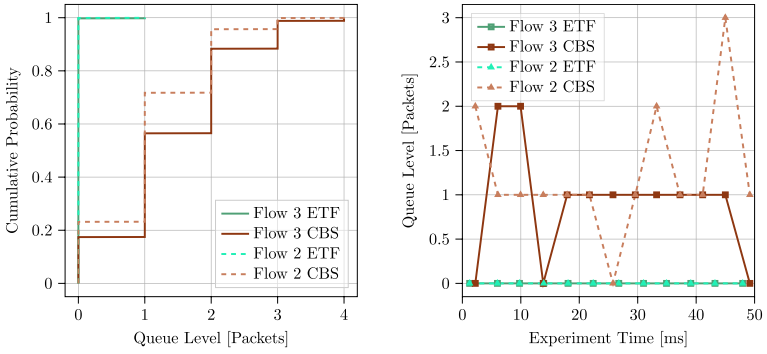
(a) ECDF **(b)** Time series

Fig. 10 End-to-end delay jitter measured for selected use-case

Figure 9 shows the measured end-to-end delay in the form of a ECDF and time series for six flows—3 TAS and 3 CBS. As expected, in Fig. 9a we see that a smaller number of hops results in lower delay. Overall, TAS using the ETF in deadline mode performs better than CBS because CBS has to wait until enough credit is built up to transmit the next packet on each hop. Such behavior is visible in Fig. 9b, where we see periodic increase and decrease of delay. On the other hand, for TAS, we see a saw tooth pattern caused by opening and closing the individual gates.

Figure 10 visualizes the end-to-end delay jitter observed in the experiments as an ECDF. Following the delay results of TAS, the delay difference between consecutive packets can be considerable. Figure 10a shows that for more than 30% of packets the jitter is larger than 100 µs. For CBS, we achieve low jitter except for a few outliers. The fluctuation of jitter is well visible in Fig. 10b.

Finally, Fig. 11 shows an ECDF of the queue levels, measured using our custom-built tool relying on *tc*, which listens on ZGW2 interface 1 and monitors queues



(a) ECDF **(b)** Time series

Fig. 11 Queue length measured on ZGW2 interface 1 for queues of Flows 2 and 3

of Flows 2 and 3. For the CBS, we see how the queues are more filled than TAS, which is possibly caused by waiting for enough credit to build. This is in some form periodic as shown in Fig. 11b. We also observe that CBS Flow 3 has a lower queue level compared to Flow 2, due to higher priority of Flow 3. Unfortunately, the tool's resolution is not high enough to catch all queue level increases/decreases. The monitoring utility only provides a periodic snapshot of the queue level every few miliseconds, where TAS cycle time is 1 ms.

6 Limitations

Even though we developed *EnGINE* with IVNs in mind, we identified few shortcomings of our approach. Our framework focuses purely on Ethernet-based solutions with the support of different bandwidths by NICs. However, networks in current vehicles are heterogeneous and support numerous bus technologies such as CAN, LIN, and FlexRay. Nevertheless, we see a shift towards the zonal architecture in vehicles, which *EnGINE* enables to evaluate. All data connected to the backbone relies on gateways that can translate from various bus systems to Ethernet. With technologies such as 10Base-T1S, Ethernet might become the dominant technology in other parts of IVNs. Similarly, we do not use specific automotive SW and HW but rather a Linux distribution and COTS HW due to $\mathbf{R}_2$ and $\mathbf{R}_{11}$–$\mathbf{R}_{13}$, which would be hard to fulfill with custom solutions. Besides, Linux with the proper configuration we use in our approach provides deterministic behavior and fulfills metrics defined in AVNU SR classes.

Furthermore, even though we improve the performance of Linux using various OS optimization techniques, it is still not the same as using a real-time OS used in VCCs and ECUs inside of vehicles or other scenarios. This affects mainly the applications running on them. Nevertheless, this challenge can be partially eliminated by understanding which delay is caused by Linux. This knowledge can then

be interpolated to the overall system performance and especially used to offer deterministic behavior for the applications. Using this approach, we can identify the worst-case scenario and expect that the overhead will be eliminated using more custom solutions. The used OS optimization helps us achieve deterministic behavior, which is crucial to the worst-case analysis.

While using tools available in Linux and building new applications, we identified new challenges regarding their performance and possible resolution, e.g., queue monitoring tool. Linux offers a large spectrum of built-in tools which provide the data we need but cannot operate in microsecond or even nanosecond order of precision. A possible solution is to make a custom application using the available kernel APIs or even rely on extended Berkeley Packet Filter (eBPF), which can run in kernel space and causes less system overhead.

Finally, there are many TSN standards that our infrastructure does not offer. Some are not yet available in Linux or are not integrated into the infrastructure, e.g., Audio Video Transport Protocol [7] or IEEE 802.1AS-Rev [6]. In case they are not integrated to Linux, it might be more challenging as the open-source community might choose a different focus instead of implementing a specific standard or you can implement own kernel module implementing a given standard. However, once available, they are easy to integrate into our infrastructure. Not every standard is of relevance in the scope of IVNs.

7 Conclusion and Future Work

We provide detailed description of *EnGINE*, a solution to repeatable, reproducible, and replicable TSN experiments with a focus on intra-vehicular networks by using COTS HW and open-source solutions. It supports various TSN standards as recommended by IEEE P.802.1DG TSN Profile [8]. Nevertheless, the usage is not limited to that and can used to evaluation to other TSN application domains. The framework comes with some challenges stemming from the use of open-source solutions during its development. Linux kernel and SW artifacts come with inherited complexity which we overcome in the implementation phase to ensure deterministic performance. However, we are still able to fulfill most of the requirements we identify in the design phase. Furthermore, we show how *EnGINE* can be used to perform IVN and TSN experiments based on an experimental use-case with detailed configuration of individual steps. We present a subset of achievable results and the inherent limitations of the framework.

We aim to integrate various realistic data sources into the infrastructure in the future exploring the options of new physical artifacts, but also using synthetic data with realistic data patterns. With this we can strengthen the realism aspect with a focus on the traffic patterns present in intra-vehicular networks. Next, even though we introduce a set of supported TSN standards, we want to extend the number of supported synchronous and asynchronous standards, e.g., IEEE 802.1Qbr and IEEE 802.1Qcr. We also want to extend our experiments and include link failures to investigate system reconfiguration times for which are purposefully designed

actions. Furthermore, the current focus is mostly on layer two functionality. In the following, we want to evaluate mechanisms on layers three and above to see how they affect performance and can be combined with deterministic guarantees provided by layer two. Finally, since EnGINE is still evolving, we aim to perform an in-depth evaluation of the framework and compare them possibly to simulation-based approaches and identify the differences while working on the items mentioned above.

Acknowledgements This work was partially funded by the EU's Horizon 2020 research and innovation programme (project SLICES-SC, 101008468), the Bavarian Ministry of Economic Affairs, Regional Development and Energy (project 6G Future Lab Bavaria), and the German Federal Ministry of Education and Research (16KISK001K).

Funding Open Access funding enabled and organized by Projekt DEAL.

References

1. Zeng, W., Khalid, M.A.S., Chowdhury, S.: In-vehicle networks outlook: achievements and challenges. IEEE Commun. Surv. Tutor. **18**(3), 1552–1571 (2016)
2. Sommer, S., Camek, A., Becker, K., Buckl, C., Zirkler, A., Fiege, L., Armbruster, M., Spiegelberg, G., Knoll, A.: Race: a centralized platform computer based architecture for automotive applications. In: 2013 IEEE International Electric Vehicle Conference (IEVC), pp. 1–6 (2013)
3. Rezabek, F., Bosk, M., Paul, T., Holzinger, K., Gallenmüller, S., Gonzalez, A., Kane, A., Fons, F., Haigang, Z., Carle, G., Ott, J.: EnGINE: Developing a flexible research infrastructure for reliable and scalable intra-vehicular TSN networks. In: 3rd International Workshop on High-Precision, Predictable, and Low-Latency Networking (HiPNet 2021), Izmir, Turkey (2021)
4. IEEE Standard for Local and Metropolitan Area Networks - Virtual Bridged Local Area Networks Amendment 12: Forwarding and Queuing Enhancements for Time-Sensitive Streams. pp. C1–72 (2010)
5. IEEE Standard for Local and Metropolitan Area Networks – Bridges and Bridged Networks - Amendment 25: Enhancements for Scheduled Traffic. pp. 1–57 (2016)
6. IEEE Standard for Local and Metropolitan Area Networks–Timing and Synchronization for Time-Sensitive Applications. IEEE Std 802.1AS-2020, pp. 1–421 (2020)
7. Pannell, Don: Automotive Ethernet AVB Functional and Interoperability Specification. https://avnu.org/wp-content/uploads/2014/05/Auto-Ethernet-AVB-Func-Interop-Spec_v1.6.pdf (2019) (Accessed on 06/02/2021)
8. Pannell, D., Chen, L., Dorr, J., Lo, W., Potts, M., Zinner, H., Zu, A.: Use cases—IEEE P802.1DG V0.4. https://www.ieee802.org/1/files/public/docs2019/dg-pannell-automotive-use-cases-0919-v04.pdf (2019). Accessed on 07 Jan 2021
9. Tuohy, S., Glavin, M., Hughes, C., Jones, E., Trivedi, M., Kilmartin, L.: Intra-vehicle networks: a review. IEEE Trans. Intell. Transp. Syst. **16**(2), 534–545 (2015)

10. Huang, X., Wang, P., Cheng, X., Zhou, D., Geng, Q., Yang, R.: The ApolloScape open dataset for autonomous driving and its application. IEEE Trans. Pattern Anal. Mach. Intell. **42**(10), 2702–2719 (2020)
11. IEEE Standard for Local and Metropolitan Area Networks–Audio Video Bridging (AVB) Systems (2011)
12. AS-2D2 Deterministic Ethernet and Unified Networking. Time-Triggered Ethernet (2016)
13. Farzaneh, M.H., Knoll, A.: Time-sensitive networking (TSN): an experimental setup. In: 2017 IEEE Vehicular Networking Conference, pp. 23–26. IEEE, 112017
14. Pfrommer, J., Ebner, A., Ravikumar, S., Karunakaran, B.: Open source OPC UA PubSub Over TSN for realtime industrial communication. In: Proceedings 2018 IEEE 23rd International Conference on Emerging Technologies and Factory Automation, pp. 1087–1090 (2018)
15. Zhao, L., Pop, P., Steinhorst, S.: Quantitative performance comparison of various traffic shapers in time-sensitive networking. CoRR, abs/2103.13424 (2021)
16. Walrand, J., Turner, M., Myers, R.: An architecture for in-vehicle networks. IEEE Trans. Veh. Technol. **70**(7), 6335–6342 (2021)
17. Mubeen, S., Ashjaei, M., Sjodin, M.: Holistic modeling of time sensitive networking in component-based vehicular embedded systems. In: 2019 45th Euromicro Conference on Software Engineering and Advanced Applications (SEAA), pp. 131–139, 82019
18. Kim, H.-J, Choi, M.-H., Kim, M.-H., Lee, S.: Development of an ethernet-based heuristic time-sensitive networking scheduling algorithm for real-time in-vehicle data transmission. Electronics **10**(2) (2021)
19. Leonardi, L., Lo Bello, L., Patti, G.: Performance assessment of the IEEE 802.1Qch in an automotive scenario. In: 2020 AEIT International Conference of Electrical and Electronic Technologies for Automotive, pp. 1–6. IEEE, 11182020
20. Hellmanns, D., Falk, J., Glavackij, A., Hummen, R., Kehrer, S., Durr, F.: On the performance of stream-based, class-based time-aware shaping and frame preemption in TSN. In: 2020 IEEE International Conference on Industrial Technology, pp. 298–303, Piscataway, NJ (2020). IEEE
21. Migge, J., Villanueva, J., Navet, N., Boyer, M.: Insights on the performance and configuration of AVB and TSN in automotive ethernet networks. In: 9th European Congress on Embedded Real Time Software and Systems (ERTS 2018), Toulouse, France (January 2018)
22. Varga, A.: OMNeT++. In: Wehrle, K., Güneş, M., Gross, J. (eds.) Modeling and Tools for Network Simulation, pp. 35–59. Springer, Heidelberg (2010)
23. Steinbach, T., Kenfack, H.D., Korf, F., Schmidt, T.C.: An extension of the OMNeT++ INET framework for simulating real-time ethernet with high accuracy. In: Proceedings of the 4th International ICST Conference on Simulation Tools and Techniques, SIMUTools '11, pp 375–382, Brussels, BEL (2011). ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering)
24. Falk, J., Hellmanns, D., Carabelli, B., Nayak, N., Dürr, F., Kehrer, S., Rothermel, K.: NeSTiNg: Simulating IEEE Time-sensitive Networking (TSN) in OMNeT++. In: 2019 International Conference on Networked Systems (NetSys) (March 2019)
25. Boehm, M., Ohms, J., Kumar, M., Gebauer, O., Wermser, D.: Time-sensitive software-defined networking: a unified control- plane for TSN and SDN. In: Mobile Communication - Technologies and Applications; 24. ITG-Symposium, pp. 1–6 (2019)
26. Haeberle, M., Heimgaertner, F., Loehr, H., Nayak, N., Grewe, D., Schildt, S., Menth, M.: Softwarization of automotive E/E architectures: a software-defined networking approach. In: 2020 IEEE Vehicular Networking Conference, pp. 1–8 (2020)
27. Hackel, T., Meyer, P., Korf, F., Schmidt, T.C.: Software-defined networks supporting time-sensitive in-vehicular communication. In: 2019 IEEE 89th Vehicular Technology Conference (VTC2019-Spring), pp. 1–5. IEEE (2019)
28. Häckel, T., Meyer, P., Korf, F., Schmidt, T.C.: Secure time-sensitive software-defined networking in vehicles. preprint arXiv:2201.00589 (2022)
29. Kostrzewa, A., Ernst, R.: Fast failover in ethernet-based automotive networks. In: 2020 IEEE 23rd International Symposium on Real-Time Distributed Computing (ISORC), pp. 134–139. IEEE (2020)
30. van Adrichem, N.L.M., van Asten, B.J., Kuipers, F.A.: Fast recovery in software-defined networks. In: 2014 Third European Workshop on Software Defined Networks, pp. 61–66. IEEE, 92014
31. IEEE Standard for Local and Metropolitan Area Network–Bridges and Bridged Networks. IEEE Std 802.1Q-2018, pp. 1–1993 (2018)

32. IEEE Standard for a Precision Clock Synchronization Protocol for Networked Measurement and Control Systems. IEEE Std 1588-2019, pp. 1–499 (2020)
33. Artifact Review and Badging - Current. https://www.acm.org/publications/policies/artifact-review-and-badging-current (08 2020). Accessed on 06 Mar 2021
34. Gallenmüller, S., Scholz, D., Wohlfart, F., Scheitle, Q., Emmerich, P., Carle, G.: High-performance packet processing and measurements. In: 2018 10th International Conference on Communication Systems Networks, pp. 1–8 (2018)
35. Scheitle, Q., Wählisch, M., Gasser, O., Schmidt, T.C., Carle, G.: Towards an ecosystem for reproducible research in computer networking. In: Proceedings of the Reproducibility Workshop, Reproducibility '17, pp. 5–8, New York, NY, USA (2017). Association for Computing Machinery
36. Gallenmüller, S., Wiedner, F., Naab, J., Carle, G.: Ducked tails: trimming the tail latency of(f) packet processing systems. In: 3rd International Workshop on High-Precision, Predictable, and Low-Latency Networking (HiPNet 2021), Izmir, Turkey (2021)
37. Gallenmüller, S., Scholz, D., Henning, S., Carle, G.: The pos Framework: a methodology and Toolchain for reproducible network experiments. In: The 17th International Conference on emerging Networking EXperiments and Technologies (CoNEXT '21), Munich, Germany (Virtual Event) (December 2021)
38. Bosk, M., Rezabek, F., Holzinger, K., Gonzalez, A., Kane, A., Fons, F., Haigang, Z., Carle, G., Ott, J.: Demo: environment for generic in-vehicular network experiments—engine. In: 2021 IEEE Vehicular Networking Conference (VNC), pp. 117–118 (2021)
39. Maddern, W., Pascoe, G., Linegar, C., Newman, P.: 1 Year, 1000km: The Oxford RobotCar Dataset. Int. J. Robot. Res. **36**(1), 3–15 (2017)
40. Pfaff, B., Pettit, J., Koponen, T., Jackson, E.J., Zhou, A., Rajahalme, J., Gross, J., Wang, A., Stringer, J., Shelar, P., Amidon, K., Casado, M.: The design and implementation of open VSwitch. In: Proceedings of the 12th USENIX Conference on Networked Systems Design and Implementation, NSDI'15, pp. 117–130, USA (2015). USENIX Association
41. Zhou, Z., Lee, J., Berger, M.S., Park, S., Yan, Y.: Simulating TSN traffic scheduling and shaping for future automotive ethernet. J. Commun. Netw. **23**(1), 53–62 (2021)
42. Emmerich, P., Gallenmüller, S., Raumer, D., Wohlfart, F., Carle, G.: MoonGen: a scriptable high-speed packet generator. In: Internet Measurement Conference 2015 (IMC'15), Tokyo, Japan (October 2015)
43. Heursch, A.C., Grambow, D., Horstkotte, A., Rzehak, H.: Steps towards a fully preemptable linux kernel. Memory **48**, 31 (2003)
44. Love, R.: Kernel Korner: CPU affinity. Linux J. **2003**(111), 8 (2003)
45. Delgado, R., Choi, B.W.: New insights into the real-time performance of a multicore processor. IEEE Access **8**, 186199–186211 (2020)
46. Marr, D.T., Binns, F., Hill, D.L., Hinton, G., Koufaty, D.A., Miller, J.A., Upton, M.: Hyper-threading technology architecture and microarchitecture. Intel Technol. J. **6**(1) (2002)
47. Charles, J., Jassi, P., Ananth, N.S., Sadat, A., Fedorova, A.: Evaluation of the Intel® Core™ i7 Turbo Boost Feature. In: IEEE International Symposium on Workload Characterization (IISWC), pp. 188–197. IEEE (2009)
48. Brodowski, D., Golde, N., Wysocki, R.J., Kumar, V.: CPU frequency and voltage scaling code in the Linux (tm) kernel. Linux kernel documentation, p. 66 (2013)

Filip Rezabek received his Master's (2020) in Communications Engineering from the TUM. He joined the Chair of Network Architectures and Services as a Ph.D. student in the same year. His main interests are network security, applied and threshold cryptography, and distributed systems resilience and robustness. Besides, he is active in TSN with focus on intra-vehicular networks and smart manufacturing. For both areas he focuses on aspects of reproducible experiments.

Marcin Bosk is a researcher and a Ph.D. student at the Chair of Connected Mobility at TUM. He obtained his Bachelor's (2018) and Master's (2020) degrees in Computer Engineering at TU Berlin. His research is focused on Time-Sensitive and In-Vehicular networks, with special interest on novel approaches to their realization. He is also interested in the 5G and beyond mobile network architecture, especially considering network slicing.

Thomas Paul received his Master's (2019) and Bachelor's (2016) degrees from the Department of Informatics at TUM. He joined the Chair of Connected Mobility already in 2015, building up and maintaining network research infrastructure. His interests include high-performance networking, compute offloading to smart NICs, building TSN testbeds, and Internet measurements.

Kilian Holzinger is a Research Associate and Ph.D. student at the Chair of Network Architectures and Services at TUM. He studied at TUM and at 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.

Sebastian Gallenmüller received his Ph.D. in 2021 from TUM where he works as a PostDoc at the Chair for Network Architectures and Services. His main research interests are programmable packet processing systems and testbeds for reproducible network experiments. The main focus of his investigations are the performance analysis and modeling of packet processing systems.

Angela Gonzalez studied Telecommunications engineering in UVIGO, and received a Master's degree in Electronics Engineering Systems from UPM. At present, she is with Huawei Technologies in the Applied Network Technology Lab of Munich Research Center focusing on HW accelerator design for automotive networking solutions, and pursuing a Ph.D. together with UPC.

Abdoul Kane received his Master's degree (2013) in Electrical Engineering with a focus on embedded electronics from the ESCPE Lyon, France. His research is focused on design of safe in-car communication networks for automated driving. He currently works at Huawei Technology's Munich Research Center as Principal Functional Safety Researcher. Previously, he worked at Infineon Technologies as an Application and Concept Engineer for automotive microcontrollers.

Francesc Fons received his Bachelor's (1995), Master's (2001), and Ph.D. (2012) degrees from the URV, Tarragona, Spain. His professional career focus is on the automotive electronics industry, working on R&D in the fields of embedded software, systems, hardware, and networks. At present, he is with Huawei Technologies, where he has the role of Chief Automotive In-Vehicle Network Researcher in the Huawei Munich Research Center.

Zhang Haigang has focused his career on the cybersecurity industry, working in R &D in embedded software, systems, and cybersecurity. Currently working for Huawei Technologies Co., Ltd., as the product line representative of the Network Application Technology Lab of Huawei Munich Research Center.

Georg Carle is a professor at the Department of Informatics at TUM since 2008, holding the Chair of Network Architectures and Services. He studied at University of Stuttgart, Brunel University, London, and École Nationale Supérieure des Télécommunications, Paris. He received his Ph.D. at University of Karlsruhe (1996), and worked as a postdoctoral scientist at Institut Eurecom, France, at the Fraunhofer Institute for Open Communication Systems, Berlin. He was professor at the University of Tübingen (2003-2008).

Jörg Ott holds the Chair of Connected Mobility at the TUM Department of Informatics since August 2015. He holds a Ph.D. (1997) and a diploma in Computer Science (1991) from TU Berlin and a diploma in Industrial Engineering from TFH Berlin (1995). His research interests are in network architectures, (transport) protocols, and algorithms for connecting mobile nodes to the Internet and to each other, covering the spectrum from delay-tolerant to real-time networking.

Authors and Affiliations

Filip Rezabek[1] · Marcin Bosk[1] · Thomas Paul[1] · Kilian Holzinger[1] · Sebastian Gallenmüller[1] · Angela Gonzalez[2] · Abdoul Kane[2] · Francesc Fons[2] · Zhang Haigang[2] · Georg Carle[1] · Jörg Ott[1]

✉ Filip Rezabek
rezabek@in.tum.de

Marcin Bosk
bosk@in.tum.de

Thomas Paul
paulth@in.tum.de

Kilian Holzinger
holzingk@in.tum.de

Sebastian Gallenmüller
gallenmu@in.tum.de

Angela Gonzalez
angela.gonzalez.marino@huawei.com

Abdoul Kane
abdoul.aziz.kane@huawei.com

Francesc Fons
francesc.fons@huawei.com

Zhang Haigang
zhanghaigang@huawei.com

Georg Carle
carle@in.tum.de

Jörg Ott
ott@in.tum.de

[1] Department of Informatics, Technical University of Munich, Munich, Germany

[2] Huawei Technologies Düsseldorf GmbH, Düsseldorf, Germany