# EnGINE: Developing a Flexible Research Infrastructure for Reliable and Scalable Intra-Vehicular TSN 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], and Jörg Ott[1]

[1]Department of Informatics, Technical University of Munich, Germany
[2]Huawei Technologies Düsseldorf GmbH, Germany
[1]{rezabek | bosk | paulth | holzingk | gallenmu | carle | ott}@in.tum.de,
[2]{angela.gonzalez.marino | abdoul.aziz.kane | francesc.fons | zhanghaigang}@huawei.com

***Abstract*—Driver assistance, self-driving, and multimedia systems have two common implications: increasing demand on network bandwidth and the need for more powerful computation nodes. As a result, intra-vehicular networks (IVNs) change their layout. They are built around central nodes connected to the rest of the vehicle via Ethernet. The usage of Ethernet presents a challenge, as it lacks support for deterministic behavior by design. The solution is found within the IEEE Time-Sensitive Networking (TSN) standards, introducing real-time, low-latency, and deterministic communication into the Ethernet ecosystem.**

**These new networked systems need to be thoroughly evaluated with IVN requirements in mind. To assess numerous configurations of IVN setups, in this work, we introduce a novel *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, which is not possible to run using proprietary solutions. *EnGINE* is based exclusively on commercial off-the-shelf components and is orchestrated by a flexible Ansible framework. This approach allows us to configure various topologies emulating realistic IVNs behavior, which is challenging using simulations. Based on available related work, we further address the challenges found in the IVNs. We derive additional requirements for experiments in the TSN domain and present our approach to fulfill them in an experimental setting. We believe that *EnGINE* provides the ideal environment for TSN network experiments.**

## I. 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 Ethernet does not, by design, offer deterministic behavior, the Time-Sensitive Networking (TSN) family of standards provides real-time guarantees to Ethernet [13].

Performance and capabilities of TSN have been a research subject in recent years with most being 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. Therefore, we introduce a solution that combines simulations' advantages while deployed on a physical topology containing machines emulating zonal gateways (ZGWs) and vehicle control computers (VCCs). This solution aims to evaluate new generations of IVNs, study the impact of growing data volume on application and network performance, and determine suitable network component distribution and interconnections in an automated manner.

Our solution forms an **En**vironment for **G**eneric **I**n-vehicular **N**etwork **E**xperiments, shortly *EnGINE*. The framework relies on the Linux networking stack offering various configurations for queuing disciplines and TSN capable commercial off-the-shelf (COTS) network interface cards (NICs). In *EnGINE* we initially focus on 802.1Qav [17], 802.1Qbv [16], and 802.1AS [19] standards with potential for extension and inclusion of higher-layer TSN capabilities.

To manage the infrastructure, we use an orchestration tool built in *Ansible* [1]. 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.

The metrics we assess and the 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 stream reservation classes and their prioritization [27]. The used TSN standards follow the recommendation of IEEE P802.1DG TSN Profile for Automotive In-Vehicle Ethernet Communications [28].

This paper presents our approach to building a configurable and flexible infrastructure that fulfills unique IVN require-

ments. 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 for conducting various network experiments to achieve deterministic behavior using Ethernet. Finally, we introduce use-cases that can be tested and show a sample configuration of *EnGINE*.

## II. Background & Related Work

As introduced in [37, 41], 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 [12]. 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 (AVB) [18], now known as the TSN working group [13], and TTEthernet [3], are proposed introducing deterministic behavior to Ethernet.

In recent years, we see various activities in this domain focused on the evaluation of individual standards on commodity or proprietary hardware [8, 30], modeling of TSN standards [6, 42, 40, 35, 26], and simulations [20, 22, 11, 25]. Unfortunately, many publications evaluating performance on physical devices rely on custom hardware [8] or use simple setups containing only a few nodes [30]. On the other hand, simulations introduce setups with a large number of nodes and data flows, as shown in [25], which uses Real-Time at Work (RTaW) Pegase commercial solution [31]. Similarly, an open-source simulator OMNeT++ [39] is used in other works [20, 22, 23] where two significant plugins offering TSN are considered, Core4Inet [33] and NeSTiNg [7].

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 [4, 10]. 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 in less than $100\,\mathrm{ms}$ or even $50\,\mathrm{ms}$, which might not be possible with traditional link-layer protocols [21, 38].

### A. TSN standards

To achieve the desired latency, we utilize Linux implementations of synchronous TSN standards. We focus on 802.1Qav and 802.1Qbv standards as considered in the P802.1DG TSN Profile for Automotive In-Vehicle Ethernet Communications [28]. 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.

**802.1Qbv** [16] Traffic Scheduling, also known as Traffic Priority (TAPRIO). A part of the 802.1Q-2018 [15] standard as "Enhancements for scheduled traffic". It provides support for synchronized scheduling of multiple traffic classes on a single interface. The traffic flow is controlled by gates that operate according to a cycle determined by the system configuration.

**802.1Qav** [17] Traffic Scheduling. A part of the 802.1Q-2018 [15] standard as "Credit-based shaper" (CBS) algorithm. It protects allocated bandwidth for each allocated SR class using a scheduling system based on credits, where transmission is allowed only when the collected credit is $\geq 0$.

**IEEE 1588** [14] 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.

**802.1AS** [19] 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 the PTP protocol is that the messages are only exchanged at layer 2 (using IEEE 802.1 MAC).

## III. Analysis

In many cases, an independent reproduction and verification of results is not an easy task. Even though ACM Policy considers reproducibility as a three-stage process [2], its adoption is still in the early stages. Reproducible research in the domain of computer networking has been a continuous activity [5, 9, 32]. 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, we believe the functionality can be extended for any real-time sensitive domains.

To achieve this, we identify a set of requirements **R** which the framework should fulfill in order to handle various experiments relevant in the TSN domain:

- $\mathbf{R}_1$ **Repeatability** – experiments can be easily repeated using the same setup by our group [2]
- $\mathbf{R}_2$ **Reproducibility** – experiments can be easily reproduced by our group and external parties using the same setup [2]
- $\mathbf{R}_3$ **Replicability** – experiments can be easily reproduced by our group and external parties using different setup offering same capabilities [2]
- $\mathbf{R}_4$ **Configurability** – choice of experiments and their parameters can be easily configured
- $\mathbf{R}_5$ **Autonomy** – experiments run without human interaction
- $\mathbf{R}_6$ **Interpretability** – generated artifacts can be analyzed and explained
- $\mathbf{R}_7$ **Realism** – works with real-world traffic patterns
- $\mathbf{R}_8$ **Scalability** – the network can handle large amount of traffic and various number of nodes
- $\mathbf{R}_9$ **Reliability** – the system can handle HW malfunctions
- $\mathbf{R}_{10}$ **Diversity** – handles a variety of input data formats

$\mathbf{R}_{11}$ **Affordability/Accessibility** – the framework does not rely on proprietary solutions which might be less accessible/affordable to other groups

$\mathbf{R}_{12}$ **Openness** – framework is built using open-source and easily accessible solutions

$\mathbf{R}_{13}$ **Updateability/Upgradeability** – the components of infrastructure can be easily updated or upgraded to satisfy new requirements

Requirements $\mathbf{R}_1$-$\mathbf{R}_3$ cover the focus on TSN infrastructure. $\mathbf{R}_4$-$\mathbf{R}_6$ are relevant from the usability and experiment preparation perspectives. This covers experiment configuration, description, and autonomous execution, as well as interpretation 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 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 [12], we can see an overview of such traffic streams and various data sources which motivates $\mathbf{R}_{10}$. 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. In the domain of IVN which offers real-time guarantees, $\mathbf{R}_9$ shall offer capabilities to emulate malfunctions on various levels in the infrastructure in order to deliver low Failure In Time 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 [2]. Similarly, proprietary solutions make upgrades to the latest technologies financially demanding.

We also identify additional requirements which were not selected as the primary focus. Current IVNs are heterogeneous as they contain various network technologies, such as CAN, LIN, MOST, FlexRay, and Ethernet. We did not consider other technologies and focus purely on Ethernet, which is a backbone of modern IVNs. Other solutions might be present in other parts of the network. These solutions may be interconnected via a gateway, which can translate to Ethernet [10].

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.

## IV. Design

*EnGINE* aims to provide an all-in-one solution for reproducible experimentation of in-vehicular networks. Based on the analysis performed in Section III, the final implementation of the architecture has to fulfill the set of requirements $\mathbf{R}$.

As shown in Fig. 1, 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.

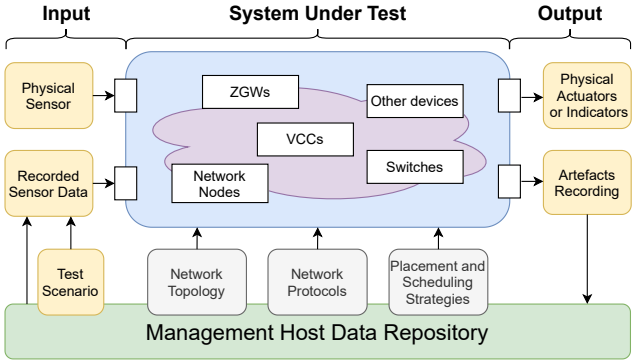



Figure 1: Overview of experiment components

Table I: 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 | $6 \times 1$ GbE Intel I210$^{\dagger}$<br>$4 \times 1$ GbE Intel I350$^{\ddagger}$<br>$2 \times 10$ GbE Intel X552 | $4 \times 1$ GbE Intel I210$^{\dagger}$<br>*or*<br>$1 \times 2.5$ GbE Intel I225$^{\dagger}$ |

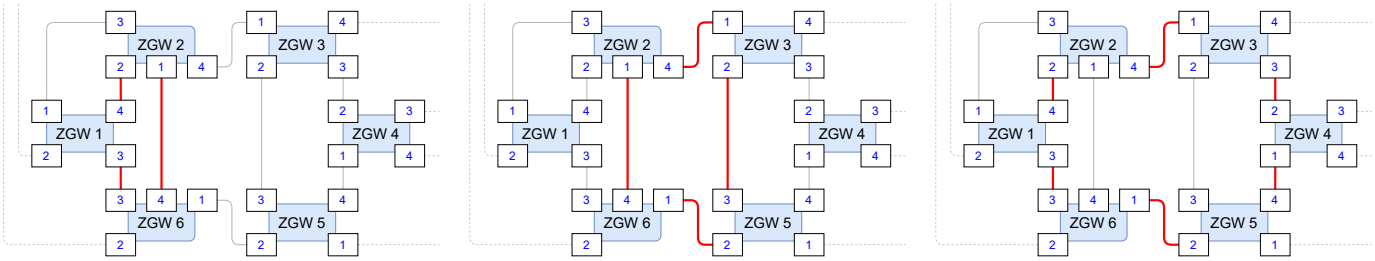
$^{\dagger}$802.1Qav, Qbv, AS, $^{\ddagger}$802.1AS

### A. Architecture

We base *EnGINE* on COTS hardware. It comprises twelve ZGWs and three VCCs. The hardware configuration of each type is shown in Tab. I. 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 support of various TSN standards, as shown in Tab. I. However, to cope with the always increasing throughput requirement, we also use Intel® X552, which is a 10GbE NIC without additional TSN support. Nevertheless, it can be used to evaluate the impact of a single non-TSN hop with remaining hops supporting at least some TSN standards.

The VCCs and ZGWs are interconnected using the network adapters mentioned in Tab. I. 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 3, 4, or 6 machines placed in a ring structure as presented in Figs. 2a to 2c respectively. With each node being equivalent to a zonal gateway (ZGW), these correspond to networks found in low-, mid-, and high-end vehicles [41]. Shown configurations can be considered a part of the same vehicle platform as well. This configuration flexibility and additional availability of higher-bandwidth $2.5\,\mathrm{Gbit/s}$ and $10\,\mathrm{Gbit/s}$ connections




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

Figure 2: Sample configurations for various IVN complexities. Red lines show links used in the respective configurations.

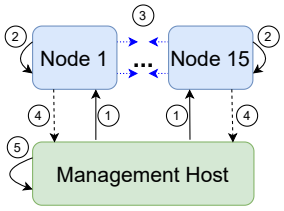
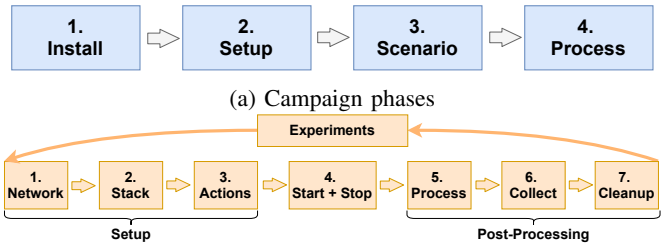



Figure 3: Communication workflow




(a) Campaign phases

(b) Individual experiment steps

Figure 4: Experiment campaign overview

satisfies requirement $\mathbf{R}_8$. Besides, AVNU alliance recommends for stream reservation classes and their corresponding metrics various topologies containing up to 7 hops [27], which are also possible within the infrastructure.

### B. Configuration and Management

To manage and configure the infrastructure, we build a custom tool using Ansible [1], an open-source configuration management software. Ansible is a descriptive language based on YAML and Jinja templates and is idempotent. 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 SSH to experiment nodes, and executes individual playbook tasks. Fig. 3 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. 4a. In the **install** phase, the nodes required for the campaign are allocated and booted with the operating system of preference. For this step, we utilize the plain orchestration service (*pos*) [9].

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 $\mathbf{R}_{13}$.

In the third phase, **scenario**, the individual experiments are conducted. Each experiment has to go through seven steps as described in Fig. 4b. 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. An example of topology can be seen in Fig. 2. The paths are placed on the network using *Open vSwitch* [29] 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 (TSN) traffic shaper of preference using Linux queuing disciplines configuration tool called *traffic control (tc)*. To note, we have granular control over the configuration of the ports and can configure each individually. Besides, each node has its own physical hardware clock, which is reflected in the PTP configuration using *linuxptp* [34]. 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 $\mathbf{R}_9$. Finally, the experiment is ***started*** and then ***stopped*** after a configured timeout. After each experiment, the generated artifacts are ***processed*** on the individual node, ***collected*** 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 final post-**processing** phase starts. Post-processing can be done either on individual or selected experiments at once to 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 the 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 which satisfies the 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 [12, 24] or synthetic data corresponding to traffic patterns [43, 31, 21]. With this approach, we can emulate various traffic patterns present in the vehicular networks and correspond to data sources available in the market. The corresponding traffic patterns can be generated using traffic generators such as *Iperf3*, *send_udp* [36], or *MoonGen* [9]. The precision of generated traffic is limited by the CPU processing, as well as the Linux scheduler. We mitigate these limitations using CPU isolation and affinity. 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 802.1AS standard, we can achieve high accuracy and precision of packet timestamps using HW timestamping.

## V. Capabilities & Limitations

*EnGINE* supports experiments ranging from small deployments of just two nodes and a single traffic flow, up to thirteen nodes and flow scenarios. We introduce a selection of the use-cases to show the configuration and capabilities of our framework. These evaluations are performed in a network of interconnected VCCs and ZGWs as shown in Fig. 5. The use-case we cover in detail comprises a single data flow, e.g. LIDAR, with a path over a single hop to a VCC as a sink. To investigate this use-case, we define an experiment campaign using a configuration of five individual YAML files (*00-nodes*, *01-network*, *02-stacks*, *03-actions*, *04-experiments*).

Starting with *00-nodes.yml*, Lis. 1, we define the nodes and their mappings used during the campaign. In Fig. 5 we show how this use-case node configuration maps to the full topology of the *EnGINE*. The remaining nodes can be used for other experiment campaigns in parallel.

Listing 1: 00-nodes.yml sample node mappings

```
---
nodes:
  - zgw5 # Source as shown in Fig. 5
  - zgw4 # Hop as shown in Fig. 5
  - vcc1 # Sink as shown in Fig. 5
node_mapping:
  node-1: zgw5
  node-2: zgw4
  node-3: vcc1
```

The defined nodes are used in the **install** phase to boot them with a predefined operating system. Before proceeding to **setup**, we need to specify network flows and TSN configuration in *01-network.yml*, Lis. 2. We identified a configuration abstraction to have sufficient control over Linux ETF, TAPRIO, and CBS qdisc 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. In the **setup** phase, also PTP is configured on all network ports specified in the corresponding configuration file. After roughly 180 s (depending on the HW performance), the nodes are ready to start with the preparation of individual applications.

Listing 2: 01-network.yml sample network configuration

```
---
network:
  net-1:
    tsn:
      tsn-1: ["node-1"]
      tsn-2: ["node-2:3"]
    flows:
      1: ":node-1:4,1:node-2:"
      2: ":node-1:4,1:node-2:3,4:node-3:"
tsnconfigs:
  tsn-1:
    name: ETF Strict and Deadline mode
    taprio: {}
    queues:
      1: { mode: etf, prio: [3], delta: 500000,
           offload: yes }
      2: { mode: etf, prio: [2], delta: 500000,
           offload: yes, deadline: yes }
      3: { mode: be, prio: ['*'] }
  tsn-2:
    ...
```

We define the applications used in a given experiment in *02-stacks.yml*, Lis. 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. We can specify how much data is transmitted over the network. Traffic volume also determines the runtime of the experiment. 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. Individual applications can terminate either after a timeout or upon completion. To successfully complete an experiment, all applications must terminate gracefully. 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.

Listing 3: 02-stacks.yml sample stack configuration

```
---
stacks:
  stack-1:
    name: UDP one way measurements BE-BE
    services:
      node-1:
        - { name: send_udp, level: 1, signal: yes,
            flow: 2 }
      node-3:
        - { name: tcpdump, level: 0, flow: 2,
            num_packets: 10000 }
```

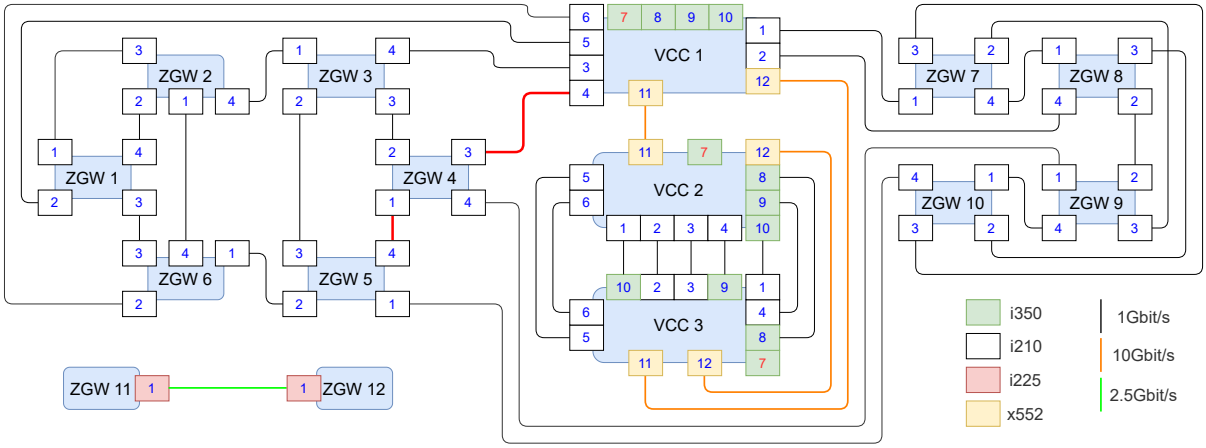



Figure 5: EnGINE network overview with connections for presented use-case marked in red

```
    protocols: {}
stack-2:
  ...
```

The main logic is contained in *04-experiments.yml*, Lis. 4, where configuration information from the other files is referenced 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 4: 04-experiment.yml sample experiment configuration

```
---
experiments:
  - { network: 'net-1', stack: 'stack-1', action:
      'action-1', signal: yes, timeout: 120,
      name: '1-hop_lidar-etf' }
  ...
```

After an experiment is successfully completed, the post-**processing** phase starts with processing results on the node before it is copied onto the management host. 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' hardware timestamping capabilities.

Other use-cases we can investigate using *EnGINE* are topologies with more than seven hops, various traffic patterns matching real data sources, or network reliability. To asses how the system behaves in case of malfunctions, we use the *03-actions.yml* introducing pre-defined failures to the system.

Even though we focus on IVNs, we identified few shortcomings of our approach. *EnGINE* 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. Neverheless, we see a shift to the zonal architecture in vehicles. 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 intra-vehicular networks. Similarly, we do not use specific automotive software and hardware but rather a Linux distribution and COTS 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, 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, i.e., AVTP and 802.1AS-Rev. The first mention might be challenging as the open-source community might choose a different focus instead of implementing a specific standard. However, once available, they are easy to integrate into our infrastructure. Not every standard is of relevance in our scope of work.

## VI. Conclusion & Future Work

We introduce a solution to repeatable, reproducible, and replicable TSN experiments with a focus on intra-vehicular networks by using COTS hardware and open-source solutions called *EnGINE*. It supports various TSN standards as recommended by P.802.1DG TSN Profile [28]. The framework comes with some challenges regarding the open-source nature of *EnGINE*. Linux kernel and software artifacts come with inherited complexity which we overcome in the implementation phase to ensure real-time performance.

In the future, we aim to integrate various realistic data sources into the infrastructure. To the best of our knowledge, there are currently no available data sources, which focus on the traffic patterns of intra-vehicular networks. Next, even though we introduce a set of supported TSN standards, we want to extend the number of supported standards, e.g., with 802.1AS-Rev, or 802.1Qbr. We also want to extend our experiments and include link failures in order to investigate system reconfiguration times. Furthermore, the current focus is mostly on layer two functionality. In the following, we want to evaluate solutions 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 while working on the items mentioned above.

## References


[1] *Ansible is Simple IT Automation*. https://www.ansible.com.

[2] *Artifact Review and Badging - Current*. https://www.acm.org/publications/policies/artifact-review-and-badging-current.

[3] AS-2D2 Deterministic Ethernet and Unified Networking. *Time-Triggered Ethernet*. Warrendale, PA, United States.

[4] M. Böhm et al. "Time-Sensitive Software-Defined Networking: A Unified Control-Plane for TSN and SDN". In: 2019.

[5] G. Carle. *Workshop on Models, Methods and Tools for Reproducible Network Research - Wrap-up*. 2003.

[6] F. Dürr and N. G. Nayak. "No-wait Packet Scheduling for IEEE Time-sensitive Networks (TSN)". In: *Proceedings of the 24th International Conference on Real-Time Networks and Systems*. New York, Oct. 2016, pp. 203–212.

[7] J. Falk et al. "NeSTiNg: Simulating IEEE Time-sensitive Networking (TSN) in OMNeT++". In: *2019 International Conference on Networked Systems (NetSys)*. Mar. 2019.

[8] M. H. Farzaneh and A. Knoll. "Time-sensitive networking (TSN): An experimental setup". In: *2017 IEEE Vehicular Networking Conference*. IEEE, 112017, pp. 23–26.

[9] S. Gallenmüller et al. "High-performance packet processing and measurements". In: *2018 10th International Conference on Communication Systems Networks*. 2018, pp. 1–8.

[10] M. Haeberle et al. "Softwarization of Automotive E/E Architectures: A Software-Defined Networking Approach". In: *2020 IEEE Vehicular Networking Conference*. 2020, pp. 1–8.

[11] D. Hellmanns et al. "On the Performance of Stream-based, Class-based Time-aware Shaping and Frame Preemption in TSN". In: *2020 IEEE International Conference on Industrial Technology*. Piscataway, NJ: IEEE, 2020, pp. 298–303.

[12] X. Huang et al. "The ApolloScape Open Dataset for Autonomous Driving and its Application". In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* (2020), pp. 2702–2719.

[13] *IEEE 802.1 Time-Sensitive Networking Task Group*. https://www.ieee802.org/1/pages/tsn.html.

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

[15] "IEEE Standard for Local and Metropolitan Area Network–Bridges and Bridged Networks". In: *IEEE Std 802.1Q-2018* (2018), pp. 1–1993.

[16] "IEEE Standard for Local and metropolitan area networks – Bridges and Bridged Networks - Amendment 25: Enhancements for Scheduled Traffic". In: (2016), pp. 1–57.

[17] "IEEE Standard for Local and Metropolitan Area Networks - Virtual Bridged Local Area Networks Amendment 12: Forwarding and Queuing Enhancements for Time-Sensitive Streams". In: (2010), pp. C1–72.

[18] *IEEE Standard for Local and metropolitan area networks–Audio Video Bridging (AVB) Systems*. Piscataway, NJ, USA.

[19] "IEEE Standard for Local and Metropolitan Area Networks–Timing and Synchronization for Time-Sensitive Applications". In: *IEEE Std 802.1AS-2020* (2020), pp. 1–421.

[20] H.-J. Kim et al. "Development of an Ethernet-Based Heuristic Time-Sensitive Networking Scheduling Algorithm for Real-Time In-Vehicle Data Transmission". In: *Electronics* (2021).

[21] A. Kostrzewa and R. Ernst. "Fast Failover in Ethernet-Based Automotive Networks". In: *2020 IEEE 23rd International Symposium on Real-Time Distributed Computing (ISORC)*. IEEE, 2020, pp. 134–139.

[22] L. Leonardi et al. "Performance assessment of the IEEE 802.1Qch in an automotive scenario". In: *2020 AEIT International Conference of Electrical and Electronic Technologies for Automotive*. IEEE, 11182020, pp. 1–6.

[23] H.-T. Lim et al. "Performance Analysis of the IEEE 802.1 Ethernet Audio/Video Bridging Standard". In: *Proceedings of the Fifth International Conference on Simulation Tools and Techniques*. Ed. by G. Riley et al. ACM, 2012.

[24] W. Maddern et al. "1 Year, 1000km: The Oxford RobotCar Dataset". In: *The International Journal of Robotics Research (IJRR)* (2017), pp. 3–15.

[25] J. Migge et al. "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, Jan. 2018.

[26] S. Mubeen et al. "Holistic Modeling of Time Sensitive Networking in Component-Based Vehicular Embedded Systems". In: *2019 45th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*. 82019, pp. 131–139.

[27] D. Pannell. *Automotive Ethernet AVB Functional and Interoperability Specification*. https://avnu.org/wp-content/uploads/2014/05/Auto-Ethernet-AVB-Func-Interop-Spec_v1.6.pdf.

[28] D. Pannell et al. *Use Cases - IEEE P802.1DG V0.4*. https://www.ieee802.org/1/files/public/docs2019/dg-pannell-automotive-use-cases-0919-v04.pdf. Sept. 2019.

[29] B. Pfaff et al. "The Design and Implementation of Open VSwitch". In: *Proceedings of the 12th USENIX Conference on Networked Systems Design and Implementation*. NSDI'15. USA: USENIX Association, 2015, pp. 117–130.

[30] J. Pfrommer et al. "Open Source OPC UA PubSub Over TSN for Realtime Industrial Communication". In: *Proceedings 2018 IEEE 23rd International Conference on Emerging Technologies and Factory Automation*. 2018, pp. 1087–1090.

[31] *RTaW-Pegase helps design safe and optimized critical embedded networks – RealTime-at-Work (RTaW)*. https://www.realtimeatwork.com/rtaw-pegase/.

[32] Q. Scheitle et al. "Towards an Ecosystem for Reproducible Research in Computer Networking". In: *Proceedings of the Reproducibility Workshop*. Reproducibility '17. New York, NY, USA: Association for Computing Machinery, 2017, pp. 5–8.

[33] T. Steinbach et al. "An Extension of the OMNeT++ INET Framework for Simulating Real-time Ethernet with High Accuracy". In: Mar. 2011, pp. 375–382.

[34] *The Linux PTP Project*. http://linuxptp.sourceforge.net/.

[35] D. Thiele et al. "Formal worst-case timing analysis of Ethernet TSN's time-aware and peristaltic shapers". In: *2015 IEEE Vehicular Networking Conference*. IEEE, 2015, pp. 251–258.

[36] *[TSN] Scheduled Tx Tools - Examples and Helpers for testing SO_TXTIME, and the etf and taprio qdiscs Âů GitHub*. https://gist.github.com/jeez/bd3afeff081ba64a695008dd8215866f.

[37] S. Tuohy et al. "Intra-Vehicle Networks: A Review". In: *IEEE Transactions on Intelligent Transportation Systems* (2015), pp. 534–545.

[38] N. L. van Adrichem et al. "Fast Recovery in Software-Defined Networks". In: *2014 Third European Workshop on Software Defined Networks*. IEEE, 92014, pp. 61–66.

[39] A. Varga. "OMNeT++". In: *Modeling and Tools for Network Simulation*. Ed. by K. Wehrle et al. Heidelberg: Springer, 2010, pp. 35–59.

[40] J. Walrand et al. *An Architecture for In-Vehicle Network*.

[41] W. Zeng et al. "In-Vehicle Networks Outlook: Achievements and Challenges". In: *IEEE Communications Surveys & Tutorials* (2016), pp. 1552–1571.

[42] L. Zhao et al. "Quantitative Performance Comparison of Various Traffic Shapers in Time-Sensitive Networking". In: (2021).

[43] Z. Zhou et al. "Simulating TSN traffic scheduling and shaping for future automotive Ethernet". In: *Journal of Communications and Networks* (2021), pp. 53–62.