Simulation and Practice: A Hybrid Experimentation Platform for TSN

Marcin Bosk, Filip Rezabek, Johannes Abel, Kilian Holzinger, Max Helm, Georg Carle, and Jörg Ott
TUM School of Computation, Information, and Technology, *Technical University of Munich, Germany*
[bosk | rezabek | abel | holzingk | helm | carle | ott] @in.tum.de

***Abstract*—Real-time systems rely on deterministic, reliable, and low-latency networks. Ethernet with Time Sensitive Networking (TSN) is used to enhance these systems' robustness while fulfilling their increasing requirements. Instead of introducing a single solution offering low latency, jitter, and packet loss, TSN provides a set of mechanisms that can be selectively combined for each specific use case. Having the means to assess various TSN standards and their configuration is crucial for successful deployments, with hardware (HW) infrastructure and simulators being common approaches. Each method presents challenges, which we aim to tackle with this work by unifying the experiment configuration and its deployment in respective environments. As a base, we use an open-source TSN framework called EnGINE. We extend the framework's functionality and provide a replacement for its HW deployment using the OMNeT++ simulator. A simulated environment is integrated via a translation layer that converts an EnGINE configuration into an OMNeT++ one. We provide design and implementation details and verify the functionality of our approach by running initial experiments and comparing them to previous results by the EnGINE authors. We show that simulation generally achieves lower delay and jitter due to its idealistic nature without typical system artifacts. However, some HW infrastructure and software-dependent configurations may unintentionally impact simulation results. Furthermore, we open-source our contributions enabling an easy way to configure once but evaluate twice while providing additional insights into HW and simulator deployments.**

***Index Terms*—TSN, Open-Source, Simulation, Experiments**

I. Introduction

Systems used for, e.g., autonomous driving or industrial automation require reliable and high throughput networks providing deterministic connectivity. Numerous solutions supporting their requirements rely on Ethernet. However, the technology does not by default offer bounded latency, jitter, and packet loss. Therefore, Ethernet is usually combined with a set of Time Sensitive Networking (TSN) standards[1], introducing various mechanisms supporting real-time guarantees.

The capabilities of TSN are being actively researched using Commercial off-the-Shelf (COTS) and professional Hardware (HW), as well as simulation deployments. Each approach has its advantages and disadvantages. Simulations mostly focus on rapid iteration, e.g., for algorithm evaluation at scale, ignoring the impact of various system artifacts or solution costs. Conversely, HW deployments offer insights into a real-world network and Operating System (OS) performance, with challenges regarding their scalability, price, and complexity. We omit emulation, e.g., *Mininet*[2], in this work as it does not yield additional insights compared to HW or simulation when considering scalability, realism, and interpretability [1].

TABLE I
Comparison of simulation, hardware deployment, and this work contributions. ✓ - Yes, ○ - Limited.

Approach	Availability	Reproducibility	Realism	Interpretability	Visibility	Scalability
Simulation	✓	✓	○	✓	✓	✓
Hardware	○	○	✓	○	○	○
This work	✓	✓	✓	✓	✓	✓

Table I shows a comparison of both approaches. Availability focuses on the accessibility of hardware needed for experimentation, being challenging for HW solutions and resulting in difficulties with the reproducibility of the results [2]. Yet, HW approaches better reflect the challenges faced in real deployments [1]. This comes at the price of the results' interpretability, as other system artifacts may be present. Similarly, obtaining visibility on artifact origins is difficult, especially when dedicated tools might cause unwanted overhead. Simulation further offers better scalability than HW infrastructure [1].

To combat the challenges of the HW approach, recently, new TSN experimentation frameworks were introduced [1], [2]. In recent work, we presented the *EnGINE* [2] framework offering a reproducible infrastructure for TSN HW-based experiments. *EnGINE* offers a generic and structured approach to experiments, provides documentation and results [3], including an open-source repository[3]. The approach still brings some challenges, including network scalability, insights into OS artifacts, e.g., queue levels, and detailed differentiation of overhead caused by the OS and algorithms' implementations.

To enhance the experimental and evaluation capabilities for TSN experimentation, in this work, we extend the *EnGINE* framework by the ability to support a simulation environment in place of the HW deployment. We re-use the framework's structured experiment scenario campaigns and execution flow

Marcin Bosk and Filip Rezabek contributed equally to this paper.
All links are valid as of 30 January 2023.
ISBN 978-3-903176-57-7
[1]https://www.ieee802.org/1/pages/tsn.html
[2]http://mininet.org/
[3]https://github.com/rezabfil-sec/engine-framework

so the definition of a given experiment campaign is done once but can be used for both deployments. Using this approach, we can combine the advantages of HW and simulation, eliminating their significant disadvantages and simultaneously improving the understanding of the HW-based results. In addition, we can use this knowledge to improve the simulator's realism and realize all features outlined in Table I.

To extend the *EnGINE* framework, we utilize its experiment execution workflow and ensure the compatibility of both approaches. Especially relevant are scenario definition and execution, combined with service configuration. As a second step, we design and extend the framework's functionality that translates the configuration for the simulator. Since the focus is on TSN, we decided to use the *OMNeT++* discrete event simulator [4] with its *INET Framework* [5] supporting numerous TSN standards. Lastly, we run initial experiments to compare the results based on the HW infrastructure used by *EnGINE* in [3] and the *OMNeT++* simulator. Such comparison allows for an initial assessment of the capabilities and a verification of our approach. We provide the integration, shown results, and documentation in *EnGINE*'s online repository[4].

We present the following Key Contributions (KCs):

KC1 Comparison of HW-based and simulation environments
KC2 Design and extension of EnGINE framework to support simulation alongside HW deployments
KC3 Comparison of HW and simulation capabilities

II. Background

Performing TSN experiments using COTS HW and open-source Software (SW), as well as simulation environments, requires interplay of multiple technologies. This is especially relevant if such an experimental system should achieve requirements, e.g., as outlined in [6]. In the following, we briefly introduce the relevant technologies and provide related work.

A. Time Sensitive Networking

Ethernet, by default, does not provide any guarantees for delay and jitter that may be induced by the network. To enable transmission of traffic requiring deterministic latencies, the IEEE 802.1Q family of TSN standards was introduced to Ethernet. These standards suggest packet policing and scheduling mechanisms tailored towards achieving low delay and jitter in time-sensitive systems. In this work, we focus on TSN standards that are readily available in Linux systems and can cooperate with COTS TSN HW, e.g., the Intel® I210 Network Interface Card (NIC) mentioned in [2]. In the following, we introduce the IEEE 802.1 Qav, Qbv, and AS standards being supported by the *EnGINE* framework.

1) Time-Aware Priority Shaper: The IEEE 802.1Qbv standard [7], also known as Time-Aware Shaper (TAS) or in Linux as Time Aware Priority Shaper (TAPRIO) queuing discipline (qdisc), enables low latency and low jitter deterministic packet delivery in Ethernet-based networks. The shaper fulfills these requirements by defining transmission times for packets of multiple traffic classes (TCLs) and hardware queues within one NIC. Packet scheduling is enforced using gates controlled according to a user-defined window cycle, specifying dedicated time-windows for each TCL. A packet can only be sent when the corresponding gate is open.

Generally, TAPRIO is combined with other mechanisms, e.g., the Earliest Time First (ETF) qdisc offering control over packet transmission times. ETF is usually bound to a NIC HW queue corresponding to a relevant TCL. The qdisc can be configured in a strict or a deadline mode. The strict mode precisely enforces the frame transmission time, whereas in deadline mode the packet may be dequeued before the desired transmission time. Some NICs enable hardware offload of ETF, e.g., the Intel® I210 NIC with its launch-time feature.

In Linux, the TAPRIO qdisc can be configured via the traffic control (tc) utility, including settings such as: `base-time` for schedule alignment across the network, `sched-entry` for cycle configuration, or `txtime-delay` enabling correction for delay caused by the system. Furthermore, the qdisc can be configured in TxTime mode, in which the packet transmission time is set automatically, e.g., for applications that do not support setting these times natively, or in offload mode where the NIC runs TAPRIO instead of the OS.

2) Credit-Based Shaper: The IEEE 802.1Qav standard [7], also known as Credit-Based Shaper (CBS), enables bandwidth allocation to TCLs defined as Stream Reservation classes. Their share is enforced using a system where packets are scheduled from corresponding queues according to credits available for each TCL. A frame can only be dequeued when the TCL credit is ≥ 0, assuming no other frames are currently transmitted by the NIC. With a configuration that matches the expected traffic patterns, CBS provides some guarantees for latency and jitter in policed TCLs.

When used in Linux, CBS is usually configured as child qdisc under the Multiqueue Priority (MQPRIO) qdisc. Alternatively, it can also be used alongside TAPRIO. The MQPRIO qdisc itself allows for frames of various TCLs to be mapped to their corresponding NIC queues and priorities, and then be policed by the child qdiscs accordingly. MQPRIO's use is not limited to CBS. It can also be used with qdiscs such as ETF or the default pfifo_fast, among others.

3) Precision Time Protocol: Accurate time synchronization is required to achieve the desired low latency and jitter in TSN. This can be achieved using Precision Time Protocol (PTP) introduced with IEEE 1588 [8] standard and extended by IEEE 802.1AS [9] in the form of generic Precision Time Protocol (gPTP) for TSN systems. The clocks of participating nodes synchronize with the help of PTP instances structured in a master-slave hierarchy. These instances exchange timing information that is used to calculate the clock offset and path delay between participating nodes. The obtained values are then used to synchronize the slave clock to the master clock. Furthermore, a Grandmaster Clock (GM) clock is placed on the top of the hierarchy, defining a reference time for the network to which all other clocks are synchronized.

[4]See Footnote 3

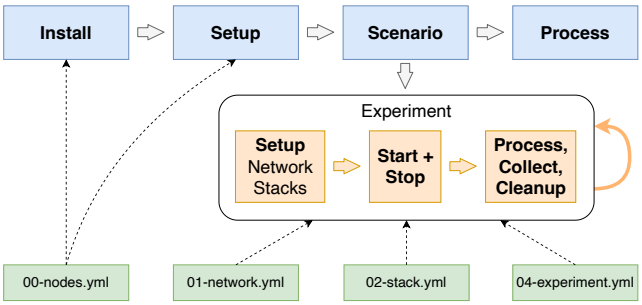

Fig. 1. *EnGINE* experiment campaign configuration and execution overview [2]

B. EnGINE Framework

The *Environment for Generic In-vehicular Network Experiments (EnGINE)* [2] is a framework tailored towards TSN experimentation with a special focus on Intra-Vehicular Networks (IVNs). It provides an open-source[5], flexible experiment orchestration tool written in *Ansible*[6] that can be used in combination with COTS HW. The framework is further part of a more generalized experimentation methodology tailored towards repeatable, replicable, and reproducible experimentation [3]. *EnGINE* employs the Linux networking stack within experiments and uses open-source SW for other required functions, e.g., *Iperf3* for traffic generation. The framework also supports extensive data collection via hardware-timestamped packet captures and provides a post-processing pipeline. The result evaluation capabilities are specifically tailored towards TSN relevant statistics such as latency or jitter, among others. *EnGINE* natively supports several TSN standards readily available for the Linux environment. These include TAPRIO and CBS qdiscs, and PTP.

The experiments in *EnGINE* are performed inside of campaigns orchestrated via a management node. Each consists of four main phases as shown in Figure 1. In the first two phases, **install** and **setup**, the nodes taking part in an experiment are prepared. In the install phase, the OS of choice is deployed on the machines, while in **setup**, all required dependencies are installed. After the **setup**, the actual experiments are performed in the **scenario** phase, covering all experiments within a campaign. Firstly, the network for each experiment is prepared, with applications used for traffic generation and data collection being readied for execution. An experiment is then started and stopped. This cycle continues until all experiments defined for the **scenario** are complete. The created results are parsed in the final, **process**, phase. The **scenario** is configured via five Ansible configuration files:

- `00-nodes.yml` define nodes for experiment campaign
- `01-network.yml` specify network topology of each individual experiment run
- `02-stacks.yml` define applications and services to be run during individual experiments
- `03-actions.yml` is currently unused [2], [3]
- `04-experiment.yml` specify campaign experiments

[5]See Footnote 3

[6]https://www.ansible.com

C. OMNeT++ Discrete Event Simulator

Open Modular Network Testbed in C++ (OMNeT++) [4] is an open-source discrete event simulation framework and environment tailored towards network simulation. It was designed with computer networks in mind. However, it can be used to simulate any type of network, examples including on-chip or queuing networks. *OMNeT++* itself provides a simulation kernel library written in C++, includes a graphical and command-line interface, and some supporting utility tools.

The functions, protocols, and other building blocks needed for network system simulation are built based on simple and compound modules. Simple modules implement basic functions, while the compound modules combine the simple modules to realize more complex functionality. The modules are usually connected via gates and channels realizing the network. The module structure is defined using a NED description language, with its functionality implemented in C++ classes. The simulation itself is configured with an INI file named after the scenario, defining parameters for all its experiments.

Domain-specific functionality is provided using frameworks that utilize the modular nature of *OMNeT++*. Notable examples include the *INET Framework* [5] used for the simulation of computer networks, or *Simu5G* [10] enabling simulation of 5G mobile networks. In this work, we focus on and utilize functionality introduced by the *INET Framework*[7]. The framework provides abstractions for many protocols of the TCP/IP stack. In its latest version, it also includes TSN standards, previously enabled only by additional frameworks such as *NeSTiNg* [11] or *Core4INET* [12]. The *INET Framework* currently includes an implementation of TSN standards such as: PTP [9], CBS [7], and TAPRIO [7], also including IEEE 802.1Qbu [7], IEEE 802.1Qcr [13], and IEEE 802.1CB [14].

D. Related Work

Over the past years, various evaluation setups that focus on TSN standards using HW and simulation deployments have been published [15]–[17]. Especially when focusing on the simulations, we see evaluation relying on the *Real-Time at Work Pegase* (*RTaW*) commercial solution[8] and also the open-source discrete simulator *OMNeT++*. Several publications rely on *RTaW* to evaluate CBS and TAPRIO standards [18], [19]. As outlined in [1], none of the available simulators include all of the TSN standards. Even if some provide many, evaluation and validation are lacking. Nevertheless, a similar limitation also applies to HW deployments.

Few publications combine both simulation and HW deployments into a single orchestration framework. Recent work mostly focuses on HW deployments [1], [15], [20] and describes various approaches to TSN evaluation. They rely on COTS HW that supports TSN standards and in some cases also special purpose HW. Their focus is on the evaluation instead of designing a larger scale platform or introducing

[7]https://inet.omnetpp.org

[8]https://www.realtimeatwork.com/rtaw-pegase/

methodology for TSN assessment, as is the case for the *EnGINE* framework [3].

Without TSN in mind, [21] is an older publication that compares constant bitrate traffic and File Transfer Protocol traffic in two simulators and an experimental testbed. The authors note that choosing the correct simulation parameters to achieve similar results as in the testbed is a hard task.

Validation of a network simulator and emulator against experiments in wireless communications is conducted in [22]. There, the main finding is that predicting delay requires careful modeling of various details such as the network stack. [23] evaluates TAS in an emulated environment utilizing virtual interfaces, showing that precise measurements become problematic when scaling the topology size.

The most similar approach to our work is introduced in [24]. The authors combine TSN hardware and simulation and present a fixed hardware setup that is represented in the simulator. The obtained results are used to provide additional realism. Nevertheless, many of the important aspects, such as scale, flexibility, and reproducibility, are not shown. This confirms our approach of combining simulation and HW infrastructure into one solution as viable. There is a variety of previous work that can be evaluated in both deployments and a multitude of additional evaluations to be done.

Based on the available related work, we identified *OMNeT++* as a suitable simulator. It provides support for various TSN standards, is open-source for academic purposes, and has a large user community.

III. Design and Implementation

To support the capabilities offered by *EnGINE* we devise and develop an abstraction of its HW and SW deployment that can be applied to *OMNeT++*. Such approach brings several benefits for TSN experimentation. It enables researchers to prepare only one configuration for both HW experiments and simulation. Further, it enables experimentation without access to TSN capable HW, while maintaining the possibility of collaboration with others that might have such access.

For the successful design and implementation of the given abstraction, we devise the following requirements:

- **R1** Use the same scenario configuration format of *EnGINE*
- **R2** Integrate seamlessly to the *EnGINE* framework
- **R3** Replicate the physical HW deployment of *EnGINE* as outlined in [2]
- **R4** Support the applications/stacks as introduced in [2]

As we are building on top of the requirements we previously defined for *EnGINE*, here we focus only on the technical aspects needed to mitigate the shortcomings of HW or simulation-only based approaches. The outlined requirements aim at enabling integration of the *OMNeT++* simulation environment with *EnGINE* compliant configuration capabilities.

To satisfy these requirements, we utilize the phase structure of *EnGINE*. The framework uses *Ansible* playbooks for installation and scenario independent setup, with a separate playbook for scenario specific setup and execution of experiments based on user-defined variable files. For the simulation,

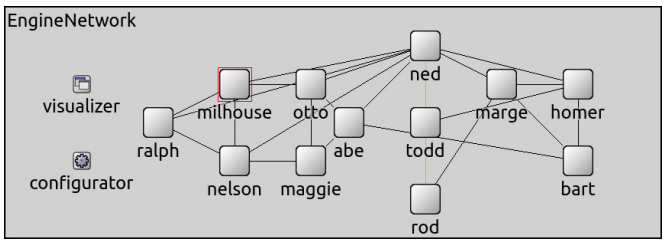

Fig. 2. *EnGINE* network topology from [2] built in *OMNeT++*

the former can be realized via default parameters used in combination with custom NED modules. In this way, we can prepare a simulation environment that corresponds to the HW deployment of the framework while fulfilling **R3**. The exact configuration of applications, network, and TSN traffic policing can be realized via scenario specific INI-files using these custom NED modules. We provide a translation framework that can transform the configuration playbooks outlined in Section II-B into the necessary INI configuration and thus fulfill **R1**. Further, we devise an implementation of traffic generation and data recording software by implementing those as *OMNeT++* modules, which combined with native *OMNeT++* statistics collection fulfills **R4** while enhancing the data collection capabilities. The entire translation framework is integrated into *EnGINE*, enabling simultaneous execution of HW emulated and simulated experiments while fulfilling **R2**. In the following, we outline the specific design and implementation choices enabling the fulfillment of all requirements.

A. Abstraction of EnGINE HW deployment in OMNeT++

The *EnGINE* HW deployment generally consists of a set of nodes differing in the quantity and quality of NICs. The wiring for the network topology remains consistent between experiments. To fulfill **R3**, an *OMNeT++* abstraction of such deployment is realized by defining appropriate modules realizing the individual nodes and their connections, as well as network modules combining them to realize the actual topology. These modules are linked via *EngineLink* channels, derived from the *INET Framework*'s *EthernetLink*. Based on [2] and aforementioned components a network topology resembling a HW setup, as shown in Figure 2, is built.

1) EnGINE Node Implementation: The custom module *EngineNode* realizes an individual node within *OMNeT++*. It is based on *INET Framework*'s *TsnDevice* to generate traffic and *TsnSwitch* to enable multiple Layer 2 flows originating on one node. A flow corresponds to a connection between two end-points and may contain traffic of multiple applications.

EngineNode combines these two modules by defining them as submodule `device` of type *EngineDevice* and `switch` of type *EngineSwitch*. The parameter `numFlows` defines the number of interfaces and links between the *EngineDevice* and *EngineSwitch* module within an *EngineNode*. Such configuration enables the setup of the per flow addresses and forwarding, as defined within the *EnGINE* framework. The two submodules `device` and `switch` are connected as needed via a custom channel per flow for which the node is an end point. The custom channel extends the *DatarateChannel*,

adding no delay and having infinite bandwidth. To enable a multi-NIC deployment of a HW node, the switch has additional interfaces that are connected to the modules gates.

The *EngineDevice* extends the functionality of *TsnDevice* and enables some of its TSN related submodules. These may be used to apply VLAN tags with the intended VLAN IDs and Priority Code Point (PCP) values for packets generated by the applications. In addition, other default settings are applied.

EngineSwitch extends *TsnSwitch* and also enables some of its TSN-related submodules. The *TsnSwitch* already includes the `gptp` and `clock` submodules facilitating the set up of time synchronization via gPTP between *EngineNode* modules. Furthermore, it contains egress traffic shaping, enabling the configuration of various shapers based on the queues associated with the interface. The settings of this queue module mimic those of the NIC queues found in the *EnGINE* framework. These include a default queue setup for interfaces, reference of correct clock submodule, or use of customized submodules enabling packet trace capture. To better mimic the capabilities of NICs used on HW *EnGINE* nodes, additional switch types extending *EngineSwitch* are introduced. These modules set the relevant number of interfaces, with their specific bandwidths and queues, for each node type available in a HW *EnGINE* deployment. Based on *EngineNode*'s parameter `type`, the respective module is chosen.

2) Network Generation and Configuration: To facilitate automatic network generation mimicking a HW deployment, we choose a three-level inheritance hierarchy for the custom network modules. Such an approach allows us to easily update the abstracted network nodes and connections in case of network topology changes. On the first level, we utilize the *EngineNetworkBase* for general definitions, independent of the actual HW nodes and their connections. The base module extends *TsnNetworkBase*, exposing its optional configuration submodules and disabling modules interfering with the setup of scenario specific flows, i.e., the setup of default routes.

Further, we extend *EngineNetworkBase* with submodules for the HW nodes that are available. *EngineNetworkNodes* defines an optional submodule of type *EngineNode* for each of the hosts outlined in [2]. The number and type of interfaces and their respective MAC addresses are set based on the information from the *EnGINE* framework's host definition files. Lastly, we define specific channels corresponding to the cabling available within the *EnGINE*'s HW deployment.

The support for TSN qdiscs is ensured by applying specific *queue* submodules to interfaces of *EngineNode*. Considering the specific structure of the *EngineNode* consisting of *EngineSwitch* and *EngineDevice*, the traffic shapers are applied on the outgoing interfaces of *EngineSwitch* used to connect with other nodes in the network. Two qdiscs are supported within our implementation, which is the same as by the *EnGINE* framework. These include *Ieee8021qTimeAwareShaper* and *Ieee8021qCreditBasedShaper*, corresponding to the IEEE 802.1Qbv and IEEE 802.1Qav standards, respectively. However, the *INET Framework* does not support the ETF qdisc. We overcome this by generating packets at precise times, which combined with comparatively low load in our considered scenarios, enables us to resemble ETF functionality.

B. Application/Stack Implementation

In the HW deployment, services abstract the use of applications during experiment runs by invoking respective commands at certain steps of the experiment preparation, run, and initial processing. These commands are contained within *Ansible* task files. In general, the service specific task file *main.yml* prepares variables and files required for the service and starts it via a custom script, enabling application control during the run of the experiment. To fulfill **R4**, for the simulation, services are realized via custom traffic generation or monitoring modules instantiated using the `app` vector of *EngineDevice*. The selected applications can be configured accordingly via scenario-specific INI configuration files.

The VLAN priority specific to each application can be applied via the bridging submodule of type *BridgingLayer*. This is realized via its submodule *StreamIdentifier* enabling setting of a `streamIdentifier` that can be used to assign packets to streams via its parameter mapping that accepts pairs of packet filters and packets. An appropriate filter has to be applied to match only the packets of those services for which the priority should be applied.

Assignment of an application to a flow can be realized via the respective remote and local IP addresses. In more detail, the `destAddress` parameter of the applications is used to select the destination endpoint of the corresponding flow and `localAddress` as the source of the flow. Similarly, the local or remote port assignment can be accomplished via the `localPort` and `destPort`, respectively.

The *EnGINE* framework stops the services and then performs initial processing via the *stop.yml*, *process.yml*, and *collect.yml* playbooks. The functionality of *stop.yml* is not required in simulation, as experiment execution is governed by the simulation kernel. Separate task files are utilized to perform service specific result processing and move result files to the persistent storage on the management node. However, parts of processing need to be adjusted to reflect the *OMNeT++* statistics collection format.

With our implementation, all services/stacks provided by the *EnGINE* framework [2] and its implementation are supported in simulation. These include, among others *gPTP* enabling time synchronization between the nodes, utilizing `gptp` and `clock` submodules of *EngineSwitch*. *tcpdump* used for packet capture recording is realized via the `pcapRecorder` module. *iperf3* used for periodic traffic generation has its client implemented with a custom *Iperf3LikePacketSource* extending *ActivePacketSource*. *send_udp* used for periodic traffic generation with ETF qdisc is integrated using the *UdpSourceApp*. Since the ETF qdisc is not available within the *INET framework*, the program's functionality can only be realized to a certain degree, i.e., the order of packets in presence of interfering traffic on the same node cannot be directly influenced.

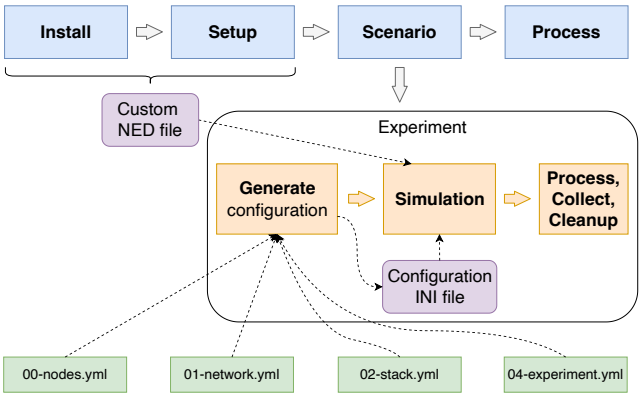

Fig. 3. Simulated *EnGINE* experiment campaign configuration and execution overview

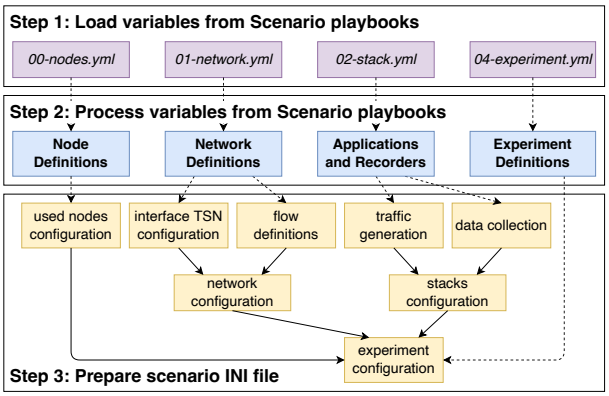

Fig. 4. Simulation configuration generation and configuration section inheritance overview

C. Experiment Execution

As outlined in Figure 3, simulated experiment execution follows the order introduced by the *EnGINE* framework. Due to the nature of *OMNeT++* some elements of the campaign phases need to be modified.

1) Installation and Setup: The allocation of nodes and installation of OS images is not required, as *OMNeT++* simulations are self-contained and only rely on a host OS where the simulator is installed. No software setup is required for the simulation since the modules will be initiated at the start of a simulation run. The default parameter values provided for the NED files are sufficient for campaign specific experiment configurations. The module instantiation happens automatically during a simulation run.

2) Scenario and Experiments: Each scenario requires specific stack and network configurations to achieve various experiment goals. This setup is realized within a single INI file, using one *Config* per experiment. The *OMNeT++* mechanism for inheriting specific INI *Configs* allows us to follow the structure of scenario specific *EnGINE* configuration files, as further explained in Section III-D. The resulting INI file enables and prepares appropriate modules needed to facilitate the network topology, traffic generation, and data collection, among others. The entire configuration needs to be performed before the experiment starts, as certain parameters cannot be dynamically set during simulation. As an example, IP addresses have to be provided in advance as the stack configuration depends on the actual network used in the experiment. We also provide additional parameters enabling similar behavior to the HW deployments, e.g., the `warmup-period` that is used to stabilize the system.

3) Process: The post-processing of artifacts in the *EnGINE* framework happens in two stages. Firstly, directly after each individual experiment, all results are pre-processed and copied to the management node. After the experimental campaign is complete, the second stage of processing occurs, where collected artifacts are parsed and evaluation plots are prepared.

In our implementation, we follow a similar approach. While *OMNeT++* supports accurate packet trace recording, we extensively utilize the simulator's native statistics, giving us a lightweight representation of the results. These statistics have the same source as the packet traces and provide identical information. These are directly extracted and parsed from *OMNeT++*'s result files using *opp_scavetool* command. Such an approach enables us to simplify some aspects of the post-processing pipeline by not having to parse packet traces without compromising accuracy. To ensure compatibility with *EnGINE*'s result evaluation capabilities, we utilize the same format for the CSV files as defined in the framework. The second stage remains exactly the same as in the original *EnGINE* implementation and can be directly used to post-process any type of experiment.

D. EnGINE Configuration Translation

OMNeT++ uses its own INI file experiment parameter configuration format. To achieve **R1**, we introduce the translation of the *EnGINE* experiment configuration into appropriate *OMNeT++* configuration files. As indicated in Figure 3, those files are generated based on playbooks shown in Section II-B and consider module descriptions introduced in Sections III-A and III-B. The translation is performed individually for each experiment campaign definition and occurs during runtime.

An overview of the configuration generation steps is shown in Figure 4. Firstly, the variables generally used by *EnGINE* during experiments are loaded via a Python script. These include the scenario-specific parameters contained within the four *Ansible* playbooks `00-nodes.yml`, `01-network.yml`, `02-stacks.yml`, and `04-experiment.yml`. The definitions are then pre-processed into a format suitable for *Jinja2* templates, where node, network, applications, and data collection recorders, and experiment definitions are derived, respectively. This step is performed to simplify the template structure as some parameter transformations are applied by the pre-processing script.

Finally, the *OMNeT++* INI configuration file is prepared using the aforementioned information and with the help of *Jinja2* templates. Several configuration sections are generated. These configurations are used as building blocks for individual experiments and help avoid redundant configuration if, e.g., multiple experiments use the same network. As indicated by arrows with dashed lines in Figure 4, the sections correspond to the structure of scenario playbooks. The figure further shows

the inheritance structure of these configurations obtained in step 3, as indicated by the arrows with solid lines. The final experiment configuration section extends all other sections and contains the definition of each individual experiment. The configuration sections for the entire scenario are contained within one INI file.

E. Integration into EnGINE Framework Execution Workflow

Supporting **R2** and the automatic nature of experiment execution in *EnGINE*, we integrate the execution of simulation runs into experiment phases, as indicated in Figure 3. Similar to the HW-based experiments, *OMNeT++* requires some preparation steps to run simulations. The setup involves installing the OS and necessary dependencies on a node dedicated to simulated experiments. Since this preparation is similar to **install** and **setup** phases of an *EnGINE* experiment, we create similar playbooks to facilitate the requirements of *OMNeT++*. These, and the simulations themselves, are executed if the simulation mode of *EnGINE* is selected. Importantly, during the **install** phase, *OMNeT++*, the supporting *INET Framework*, as well as the project containing *EnGINE*-specific module definitions and implementation are compiled. As a result, an executable binary for the project is created.

With the node being ready, the experiment preparation and execution are performed as part of the **scenario** phase. As *Ansible* variable files are available at the management node, the scenario playbook generates the scenario-specific configuration based on the scripts and templates there, subsequently moving the resulting INI file to the simulation node. The framework then sequentially executes experiments from the indicated experiment INI file section, via the executable binary created for the engine project during **install** phase. The experiments are generally run in command line mode, however, a graphical environment is also available. As described in Section III-C3, similarly to a HW *EnGINE* experiment, generated artifacts are copied after each experiment run. Finally, after a campaign is complete, the results are processed using already existing *EnGINE* post-processing capabilities.

IV. Verification Experiments and Results

With the previous work done on the *EnGINE* framework, we provided several artifacts we now use to reproduce its HW deployment and experiments in the simulation environment we introduce with this work. We utilize the following information:

- **A1** The *EnGINE* framework node definitions[9]
- **A2** The network topology as introduced in [2], [3] with additional insights from [25]
- **A3** Previously introduced experiment descriptions[10]
- **A4** Previously introduced experiment configurations[11]
- **A5** Previously introduced results[12] of [3]

Using **A1** and **A2**, we abstract the HW deployment of *EnGINE* outlined in [2], [3] into *OMNeT++*. We appropriately

[9]https://github.com/rezabfil-sec/engine-framework/tree/main/host_vars
[10]https://github.com/rezabfil-sec/engine-framework#experiments
[11]https://github.com/rezabfil-sec/engine-framework/tree/main/scenarios
[12]https://nextcloud.in.tum.de/index.php/s/sWxadG8JeJss2Sy

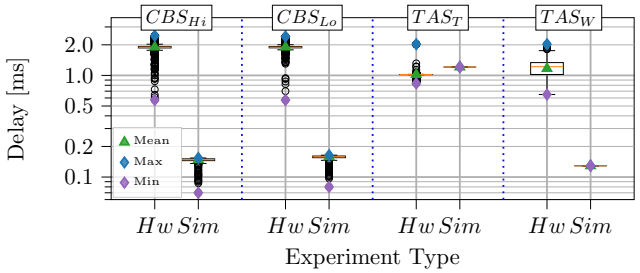

Fig. 5. Delay comparison between HW-based and simulated results

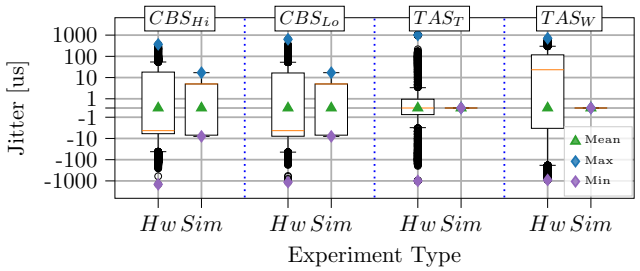

Fig. 6. Jitter comparison between HW-based and simulated results

define all network nodes as *EngineNode* modules and mimic the links between them via *EngineLink* channels. We use this network, together with **A3**, **A4**, and **A5**, to validate our simulation integration for the framework. For comparison, we utilize a small subset of the aforementioned results. We firstly focus on experiment EX_{CM2} of [3], investigating the functionality of the CBS qdisc. We further extend our validation with experiments EX_{TS-T} and EX_{TS-W} without offload of [3], focusing on the TAPRIO qdisc. We use those to verify the functionality of our solution in a line topology with seven hops and eight network nodes. The traffic is generated on the first and forwarded towards the last node in the line, with traffic shaping applied on each hop in the investigated network. To perform an accurate representation of the two experiments in simulation, we use the configurations provided via **A4** with scenarios *Figure-23-24_EX_CM2* and *Figure-25ab_EX_TS-T,EX_TS-W*. Our results are then compared against those from **A5**, which are visualized in Figures 24 and 26 of [3].

We, therefore, devise three scenarios CBS, TAS_T, and TAS_W, corresponding to, in order, the previously mentioned experiments. CBS experiment uses CBS on two queues policing traffic of two flows, with their bitrates and corresponding qdisc parameters configured for $100\,\mathrm{Mbit/s}$ each. In parallel, to saturate the link, two other flows are configured on best-effort priorities. For evaluation, we consider only the CBS policed flows for the highest and second highest priorities, in the following referred to as CBS_{Hi} and CBS_{Lo}, respectively. In contrast, both TAS experiments include only one flow. TAS_T tests larger TAPRIO window cycle of $1\,\mathrm{ms}$, with packets generated every $1\,\mathrm{ms}$. TAS_W performs a similar test with smaller TAPRIO window cycle of $100\,\mu\mathrm{s}$, with packets created according to that cycle.

Figures 5 and 6 show box plots of the delay and jitter measured in CBS and TAS experiments for both experiment types, the HW-based (Hw) from [3], as well as the corre-

sponding simulation (Sim) runs. For each individual experiment, the plots feature the measured minimum, maximum, and average values. This is in addition to the median, inter-quartile range, the extent of the data, and outliers, which are standard for a box plot, providing additional insights into the results.

For CBS_{Hi} and CBS_{Lo}, we observe similar delays within their respective experiment types. In both cases, the delay measured in Sim is significantly lower compared to Hw. While CBS in Hw shows an average delay of roughly $2.40\,\mathrm{ms}$, Sim yields an average latency of roughly $0.15\,\mathrm{ms}$. This discrepancy is a result of the processing delay currently not being modeled in our simulation environment.

Similarly, for the jitter of CBS experiments shown in Figure 6, we observe significantly lower jitter in Sim compared to Hw. Again, the flows on two different priorities are similar within their respective experiment types. While in Hw we see a maximum absolute jitter value of approx. $1400\,\mu\mathrm{s}$, for Sim this value drops significantly to just $8\,\mu\mathrm{s}$. Again, here we also see the limitation of the simulation approach, where clock drift and external influences are not fully modeled.

For the delay of TAS_T and TAS_W, shown in Figure 5, we observe significant differences comparing Hw with Sim. With TAS_T, we observe a comparable mean delay in both experiment types, with that of Sim equal to $1.20\,\mathrm{ms}$ being somewhat higher than the one in Hw of $1.00\,\mathrm{ms}$. In contrast, TAS_W shows similar behavior as seen in the CBS experiments, with Sim latency being significantly lower than that of Hw. The HW-based experiment type shows an average delay of $1.17\,\mathrm{ms}$, while simulation yields only $0.13\,\mathrm{ms}$.

Both TAS experiments let us observe an interesting case where simulation might be beneficial to determine optimal window settings for the TAPRIO qdisc. TAS_T applies an offset to the gate schedule of $200\,\mu\mathrm{s}$ per hop. This is in line with the expected processing delay per hop in the HW-based experiments, resulting in a delay increase of roughly $200\,\mu\mathrm{s}$ per hop and, therefore, an optimal configuration of TAPRIO windows in the network as described in [3]. This processing delay is not present in Sim, with packets arriving before their gate opens on every hop node. This results in the increase in delay by roughly $200\,\mu\mathrm{s}$ on each hop while the packets need to wait for their window to open. Such observations, combined with the ability to model processing delay, in the future may be used to define optimal TAPRIO windows based on varying processing abilities of different nodes.

The jitter in TAS experiments shown in Figure 6, also indicates a limitation of the simulation approach. For both TAS_T and TAS_W in Sim we see no jitter whatsoever, while Hw lets us observe absolute jitter values of up to $1000\,\mu\mathrm{s}$. This, again, indicates that certain realism aspects are omitted in the simulation approach. In summary, we observe differences between the simulated and HW-based results, showing the discrepancy between near-ideal simulation and real-world, which might also result from HW or SW implementations. The results indicate more modeling work is needed to narrow down which properties result in these unique artifacts.

V. Limitations

While our approach extends the capabilities of HW-based TSN experiments by adding additional insights via simulation, it still has several limitations. These reflect the general limitations of discrete-event simulators such as *OMNeT++* and others used to simulate computer networks. The first major limitation concerns the availability of ETF qdisc in the simulator. Currently, it is not supported by the *INET Framework*. This somewhat limits the variety of experiments involving the combination of TAPRIO and ETF, since *OMNeT++* does not differentiate between the deadline and strict modes and supports only the strict mode.

Further, the implementation of PTP in the *INET Framework* requires a pre-defined PTP hierarchy and does not support its dynamic creation. Our implementation pre-computes the same hierarchy as the Best Master Clock Algorithm for our particular setup. In *OMNeT++*, the PTP currently does not have a significant impact on the accuracy of the clocks in the network. The used clocks provide the capability to configure various types of clock drift. However, currently these features are not heavily utilized and require further investigation in order to provide a realistic clock behavior.

The *EnGINE* framework, as well as this implementation, supports only a number of TSN traffic shapers. These are limited to CBS, TAPRIO, and ETF (for *EnGINE*), dictated by the small number of corresponding qdisc implementations being available for both Linux and *OMNeT++*. Both the framework and our simulation environment can be extended by additional qdiscs if new implementations become available.

Additionally, there are some functional differences between the Linux-based applications used within HW-based *EnGINE* deployment and their corresponding *OMNeT++* modules. As an example, the *Iperf3* application may be configured to send at a non-limited rate which would saturate the remaining bandwidth of the network. Such functionality is not supported by the application module within the simulation environment. To achieve link saturation in the simulation, the unlimited bitrate flows require an additional step in the configuration, where an appropriate bitrate is set. Supported applications also need to be actively maintained. If applications are modified or new ones are added into the *EnGINE* framework, to maintain compatibility, these applications would also need to be implemented into the *OMNeT++* simulator.

VI. Conclusion & Future Work

In this work, we introduce and verify an approach enabling experimentation using both HW-based and simulated environments. We achieve this by using an open-source framework called *EnGINE* and mirroring its configuration and deployment in the open-source *OMNeT++* simulator. The mapping happens for each defined experiment scenario, allowing the same configuration to be run on hardware and in simulation. Combining both results enables the collection of additional insights from each environment without their individual limitations. The results can be used in both directions – improving

the understanding of HW-based results or validating the simulator's features, enhancing its realism. We provide a detailed description of how the functionality mapping is designed and provide a set of verification results. These results are compared with ones we previously provided with *EnGINE* and serve as initial motivation for future work possibilities.

We see potential for valuable insights into TSN standards when they can be evaluated using HW together with a simulator in the loop. Modeling differences between the two types of deployments could help to have a better understanding of how the simulation differs from real-world deployments. Such experience can also be used in reverse, where the hardware model can be applied to simulation results, e.g., for evaluation of a new TSN standard only supported in the simulator. We show that simulation achieves lower delay and jitter due to the absence of typical system artifacts.

In the future, we plan to look in more detail at the assessment of the CBS and TAPRIO qdiscs, comparing the two types of results and modeling the differences. As a part of that, we want to investigate various network calculus approaches and verify their behavior in both deployments. We plan to also evaluate additional aspects such as the PTP clock drift. In addition, the translation framework can be extended to support new tools, e.g., traffic generators or analyzers, with the potential to include real applications in the simulation environment. Last, since the solution is open-sourced, it could serve as a collaboration platform among various researchers, where some have the HW capabilities and experience, while others focus on simulation.

ACKNOWLEDGMENT

This work has been partially supported by the Federal Ministry of Education and Research of Germany (BMBF) project 6G-Life (16KISK002) and Algorand Centres of Excellence (ACE) Programme (https://www.algorand.foundation/ace).

REFERENCES

[1] M. Ulbricht, S. Senk, H. K. Nazari, H.-H. Liu, M. Reisslein, G. T. Nguyen, and F. H. Fitzek, "Tsn-flextest: Flexible tsn measurement testbed (extended version)," *arXiv preprint arXiv:2211.10413*, 2022.

[2] F. Rezabek, M. Bosk, T. Paul, K. Holzinger, S. Gallenmüller, A. Gonzalez, A. Kane, F. Fons, Z. Haigang, G. Carle, and J. Ott, "Engine: Flexible research infrastructure for reliable and scalable time sensitive networks," *Journal of Network and Systems Management*, vol. 30, no. 4, p. 74, 2022.

[3] M. Bosk, F. Rezabek, K. Holzinger, A. G. Marino, A. A. Kane, F. Fons, J. Ott, and G. Carle, "Methodology and infrastructure for tsn-based reproducible network experiments," *IEEE Access*, vol. 10, pp. 109 203–109 239, 2022.

[4] A. Varga, "OMNeT++," in *Modeling and Tools for Network Simulation*, K. Wehrle, M. Güneş, and J. Gross, Eds. Heidelberg: Springer, 2010, pp. 35–59.

[5] L. Mészáros, A. Varga, and M. Kirsche, "Inet framework," pp. 55–106, 2019. [Online]. Available: https://doi.org/10.1007/978-3-030-12842-5_2

[6] "ISO/IEC/IEEE International Standard - Information technology – Telecommunications and Information Exchange between Systems – Local and Metropolitan Area Networks – Specific Requirements – Part 1BA: Audio video bridging (AVB) Systems," *ISO/IEC/IEEE 8802-1BA First edition 2016-10-15*, pp. 1–52, 2016.

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

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

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

[10] G. Nardini, D. Sabella, G. Stea, P. Thakkar, and A. Virdis, "Simu5g–an omnet++ library for end-to-end performance evaluation of 5g networks," *IEEE Access*, vol. 8, pp. 181 176–181 191, 2020.

[11] J. Falk, D. Hellmanns, B. Carabelli, N. Nayak, F. Dürr, S. Kehrer, and K. Rothermel, "Nesting: Simulating ieee time-sensitive networking (tsn) in omnet++," in *2019 International Conference on Networked Systems (NetSys)*, 2019, pp. 1–8.

[12] T. Steinbach, H. D. Kenfack, F. Korf, and T. C. Schmidt, "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*, ser. SIMUTools '11. Brussels, BEL: ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), 2011, p. 375–382.

[13] "IEEE Standard for Local and Metropolitan Area Networks–Bridges and Bridged Networks - Amendment 34:Asynchronous Traffic Shaping," *IEEE Std 802.1Qcr-2020 (Amendment to IEEE Std 802.1Q-2018 as amended by IEEE Std 802.1Qcp-2018, IEEE Std 802.1Qcc-2018, IEEE Std 802.1Qcy-2019, and IEEE Std 802.1Qcx-2020)*, pp. 1–151, 2020.

[14] "IEEE Standard for Local and Metropolitan Area Networks–Frame Replication and Elimination for Reliability," *IEEE Std 802.1CB-2017*, pp. 1–102, 2017.

[15] M. H. Farzaneh and A. Knoll, "Time-Sensitive Networking (TSN): An Experimental Setup," in *2017 IEEE Vehicular Networking Conference*. IEEE, 112017, pp. 23–26.

[16] H.-J. Kim, M.-H. Choi, M.-H. Kim, and S. Lee, "Development of an Ethernet-Based Heuristic Time-Sensitive Networking Scheduling Algorithm for Real-Time In-Vehicle Data Transmission," *Electronics*, vol. 10, no. 2, 2021.

[17] L. Leonardi, L. Lo Bello, and G. Patti, "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.

[18] J. Migge, J. Villanueva, N. Navet, and M. Boyer, "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.

[19] C. Mauclair, M. Gutiérrez, J. Migge, and N. Navet, "Do We Really Need TSN in Next-Generation Helicopters? Insights From a Case-Study," in *2021 IEEE/AIAA 40th Digital Avionics Systems Conference*, 2021.

[20] S. Senk, M. Ulbricht, J. Acevedo, G. T. Nguyen, P. Seeling, and F. H. P. Fitzek, "Flexible measurement testbed for evaluating time-sensitive networking in industrial automation applications," in *2022 IEEE 8th International Conference on Network Softwarization (NetSoft)*, 2022, pp. 402–410.

[21] G. F. Lucio, M. Paredes-Farrera, E. Jammeh, M. Fleury, and M. J. Reed, "Opnet modeler and ns-2: Comparing the accuracy of network simulators for packet-level analysis using a network testbed," *wseas transactions on computers*, vol. 2, no. 3, pp. 700–707, 2003.

[22] S. Ivanov, A. Herms, and G. Lukas, "Experimental validation of the ns-2 wireless model using simulation, emulation, and real network," in *Communication in Distributed Systems - 15. ITG/GI Symposium*, 2007.

[23] M. Ulbricht, J. Acevedo, S. Krdoyan, and F. H. Fitzek, "Emulation vs. Reality: Hardware/Software Co-Design in Emulated and Real Time-sensitive Networks," in *26th European Wireless Conference*, 2021.

[24] J. Jiang, Y. Li, S. H. Hong, M. Yu, A. Xu, and M. Wei, "A Simulation Model for Time-sensitive Networking (TSN) with Experimental Validation," in *2019 24th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*, 2019, pp. 153–160.

[25] F. Rezabek, M. Helm, T. Leonhardt, and G. Carle, "PTP Security Measures and their Impact on Synchronization Accuracy," in *18th International Conference on Network and Service Management (CNSM 2022)*, Thessaloniki, Greece, Nov. 2022.