Demo: Environment for Generic In-vehicular Network Experiments - EnGINE

Marcin Bosk[*,1], Filip Rezabek[*,1], Kilian Holzinger[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]{bosk | rezabek | holzingk | carle | ott}@in.tum.de,
[2]{angela.gonzalez.marino | abdoul.aziz.kane | francesc.fons | zhanghaigang}@huawei.com

***Abstract*—Self driving cars bring new challenges to intra-vehicular networks (IVNs), such as increased bandwidth and the need for more powerful computation. To solve these challenges, in recent years, the IVNs started shifting towards the use of Ethernet, which with IEEE Time-Sensitive Networking standards can support deterministic behavior required for these networks. To evaluate the application of Ethernet in IVNs, we introduce novel approach called *EnGINE* - an *En*vironment for *G*eneric *I*n-vehicular *N*etworking *E*xperiments. The environment is based on commercial off-the-shelf components and open-source software. In this work we present *EnGINE* and showcase an exemplary use-case it can support.**

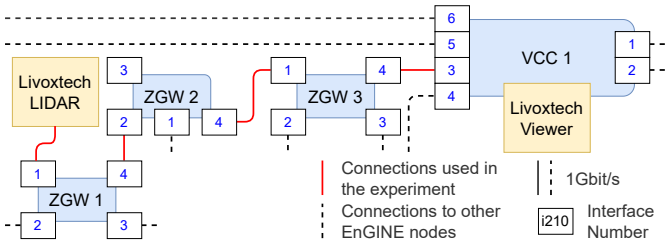

Figure 1: Excerpt of *EnGINE* network topology

I. Introduction

Autonomous driving and shared mobility are just two examples of modern trends within the automotive industry. These novelties are enabled by deterministic real-time communication in an intra-vehicular network (IVN). Due to rising demands on bandwidth, we start to see more Ethernet-based IVN solutions. Although Ethernet, by design, cannot offer deterministic behavior, with the Time-Sensitive Networking (TSN) family of standards it can provide real-time guarantees.

TSN has been a major research subject in recent years with most experiments being conducted using simulation. This approach has its advantages such as easy reproducibility and configurability. Still, it has many challenges, e.g., omission of clock-deviation and other real-deployment artifacts.

To combine the simulations' ease-of-use and behavior of real network deployments we introduce *EnGINE* [7] - an **En**vironment for **G**eneric **I**n-vehicular **N**etwork **E**xperiments. Instead of using micro-controllers [4], *EnGINE* uses PCs which emulate zonal gateways (ZGWs) and vehicle control computers (VCCs). It builds upon the Linux networking stack and its queuing disciplines (qdisc) with initial focus on 802.1Qav, 802.1Qbv, and 802.1AS TSN standards [5] unlike previous work using OpenAvnu [8]. *EnGINE* is managed using a custom-built *Ansible*-based [1] orchestration tool which allows for configuration of the network and data sources/sinks. The setup follows recommendations of the AVNU Alliance and TSN Standards outlined in IEEE P802.1DG [6]. Furthermore, the framework enables extensive monitoring that can later be used for evaluation or traffic re-play. Using *EnGINE* the experiments are conducted autonomously, and can be easily repeated and reconfigured. Thanks to the framework's flexibility, novel IVN architectures can be developed.

In this work, we show our design goals and their realization for an easily configurable and flexible IVN experiment infrastructure. Furthermore, we introduce one of the IVN use-cases and demonstrate how it can be evaluated using *EnGINE*.

[*]Marcin Bosk and Filip Rezabek contributed equally to this paper.

II. System Design

To provide a framework for IVN research, we follow the ACM Policy [2] and define additional requirements for our system: repeatability, reproducibility, and replicability. With those requirements in mind, we build *EnGINE* using commercial off-the-shelf (COTS) network interface cards (NICs). More specifically, the $1\,\mathrm{Gbit/s}$ Intel® I210 NICs, which are used to interconnect our Linux-based ZGWs and VCCs as outlined in Fig. 1. All nodes utilize standard PC components. *Open vSwitch* is used to prepare the network topology for each experiment and forward traffic.

To orchestrate *EnGINE*, we built a framework using *Ansible* [1], an idempotent, descriptive language using YAML and Jinja. Fig. 2 shows a typical workflow, where the management host remotely executes commands on a node ①. Then ② respective node runs this code and ③ interacts with others. Next, the nodes store the collected artifacts on the management host ④, where the collected artifacts are processed ⑤.

Each experiment campaign consists of four phases: **install**, **setup**, **scenario**, and **process**. In **install**, the nodes required for the campaign are allocated and booted with an operating