Time-Sensitive Autonomous Architectures

Donato Ferraro[1,2*], Luca Palazzi[1,2], Federico Gavioli[2], Michele Guzzinati[3], Andrea Bernardi[3], Benjamin Rouxel[1,2], Paolo Burgio[2] and Marco Solieri[1]

[1*]Minerva Systems, 213/B, Via Campi, Modena, 41125, Italy.
[2]Department of Physics, Informatics and Mathematics, Università degli studi di Modena e Reggio Emilia, 213/A, Via Campi, Modena, 41125, Italy.
[3]Hipert, 50, Via della Scienza, Modena, 41122, Italy.

*Corresponding author(s). E-mail(s): donato.ferraro@minervasys.tech;
Contributing authors: luca.palazzi@minervasys.tech; 224833@studenti.unimore.it; michele.guzzinati@hipert.it; andrea.bernardi@hipert.it; benjamin.rouxel@unimore.it; paolo.burgio@unimore.it; marco.solieri@minervasys.tech;

Abstract

Autonomous and software-defined vehicles (ASDVs) feature highly complex systems, coupling safety-critical and non-critical components such as infotainment. These systems require the highest connectivity, both inside the vehicle and with the outside world. An effective solution for network communication lies in Time-Sensitive Networking (TSN) which enables high-bandwidth and low-latency communications in a mixed-criticality environment. In this work, we present Time-Sensitive Autonomous Architectures (TSAA) to enable TSN in ASDVs. The software architecture is based on a hypervisor providing strong isolation and virtual access to TSN for virtual machines (VMs). TSAA latest iteration includes an autonomous car controlled by two Xilinx accelerators and a multiport TSN switch. We discuss the engineering challenges and the performance evaluation of the project demonstrator. In addition, we propose a Proof-of-Concept design of virtualized TSN to enable multiple VMs executing on a single board taking advantage of the inherent guarantees offered by TSN.

Keywords: autonomous driving, virtualization, Time-Sensitive Networking, embedded systems

1 Introduction

The increasingly high level of automation of current Autonomous Vehicles prototypes (AVs) requires unprecedented computing power, at reduced Size, Weight and Power (SWaP), but at the same time must meet the tight latency requirements of advanced control loops, providing real-time guarantees. To accomplish higher automation, we foresee that AVs will be equipped with a number of increasingly complex sensors, processors, and Electronic Control Units (ECUs), which when connected all together further exacerbate the (already critical) integration tasks of the in-vehicle network architecture, and the amount of data transiting throughout the system. Moreover, these systems involve communications with external edges (Vehicle-to-everything, V2X), such as traffic lights, intelligent pedestrian assistants, nearby vehicles (V2V), and much more.

Nowadays, the in-vehicle network uses different types of buses to communicate, such as the Controller Area Network (CAN) (Corrigan, 2016) and its derivatives, FlexRay (FlexRay Consortium, 2010), Local Interconnect Network (LIN) (Hackett, 2022), amongst others. However, these types of networks do not have a high bandwidth capacity; hence, Ethernet-based communications were introduced for both off-/in-vehicle networks (Wang et al, 2019). Since safety is one of the most important requirements in the automotive domain, it is necessary to evaluate and set all characteristics of the used communication in a worst case scenario, i.e., latencies, maximum bandwidth, jitters, packet loss rate, etc. Modern high-performance embedded ECUs, such as those powered by NVIDIA or Xilinx accelerators, can work jointly with off-vehicle cloud servers to serve this purpose. However, the key point of predictable data transmission amongst them is still an open research area. A more recent technology enabling networking capabilities for high bandwidth and low latency communication in a mixed criticality environment is named Time-Sensitive Networking (TSN) (Farkas et al, 2018). TSN is a set of standards that extend the Ethernet protocol by focusing on transmission time guarantees (Finn, 2022). Ethernet TSN communications can be achieved with specific components providing switching and regulation to frames traffic, such as silicon components or Field Programmable Gate Array (FPGA) based switches. The first challenge of our interest is *applicability of TSN-based networking to complex applications like autonomous cyber-physical systems.*

Another fast-growing automotive trend aims to provide unprecedented flexibility and ease of development to hardware/software architecture. A Service-Oriented Architecture, on the one hand, can deliver a Software-Defined Vehicle (SDV) that can evolve and adapt along with time and the user's needs. The edge-cloud continuity first enables a convenient development and testing

environment during the product conception, and then allow computing distribution along vehicle and infrastructure during the product operation Kane et al (2022); Andreozzi and Shirasat (2022); SOAFEE (2022). The second challenge of our interest is *enabling software-definedness to TSN-networked hardware/software architectures*.

This paper introduces Time-Sensitive Autonomous Architecture (TSAA), the next-generation software multi-zonal architecture that leverages TSN protocols to manage mixed-criticality system data flows in software-defined vehicles.

Fig. 1 A picture of *Maserati Quattroporte* used for the experiments.

Our first contribution is the description of the TSAA concepts and its applicability in real-time applications. Our case-study is based on a real self-driving car that is capable of accomplishing safety-critical L4 functionality, i.e., following a predetermined trajectory in a safe manner, even in the presence of high network traffic/interference. This case-study was instantiated, tested, and evaluated directly on-field in our *Maserati Quattroporte*, a luxury sports car targeted by this project, shown in Figure 1.

Our second contribution is a Proof-of-Concept (PoC) of *Virtual TSN (VTSN)*, a software solution that brings TSN switching capabilities to a virtualized Ethernet endpoint. Flexibility and scalability of the hardware/software are provided to a hypervisor-based environment where virtual machines can enjoy TSN and its quality-of-service. With VTSN, a single Ethernet port is managed by a dedicated VM executing the TSN scheduling algorithm. This dedicated VM is used as a proxy by the other ones for incoming and outgoing traffic. An early prototype of VTSN prototype is evaluated to demonstrate feasibility and validity of the approach.

This paper is organized as follows: in Section 2, we review the literature of existing technologies in the realm of virtualization and timing sensitive networks. In Section 3, we detail the TSAA future prototype by describing and motivating our choices. In Section 4, we introduce a PoC as well as a feasibility analysis of the VTSN solution. In Section 5, we explore the engineered solutions taken for the case-study of the TSAA applicability to a self-driving vehicle

capable of following a predefined trajectory even in the presence of interference. In Section 6, we present the prototype in its current state. Finally, in Section 7 we summarize the experience we gained by engineering the autonomous car, and open future research directions.

2 Related Works

Systems designed with virtualization reported great advantages related to security, cost, reliability, availability, and adaptability, making it a valid choice with remarkable performance (Obasuyi and Sari, 2015). In recent years, paravirtualization has exhibited higher performance compared to full virtualization, where all the hardware is emulated, like in Qemu (Motika and Weiss, 2012; Fayyad et al, 2013). In addition to partitioning computing resources, paravirtualization enables multiple applications to gain access to the hardware resources on the host machine (Obasuyi and Sari, 2015).

One hardware resource concerns connectivity, as the scarcity of communication connectors raises new challenges for hypervisors: they need to also act as a transparent proxy between the network and the OSes operating in a virtualized environment. Techniques have therefore been proposed in order to enable time-sharing mechanisms for shared hardware resources across all guest machines through different protocols.

The most well-known protocol used in system automation is the Controller Area Network (CAN) (Gergeleit and Streich, 1994), for which an extension for virtual environments vCAN has been proposed with great results (Herber et al, 2014; Breaban et al, 2016). The field-bus protocol EtherCAT (Jansen and Buttner, 2004) is another example. This protocol is suitable for both hard and soft real-time computing requirements in automation technology. It has been used in a real-time virtualized context for industrial automation, and EtherCAT has proven to be a suitable choice for real-time control systems with guaranteed performance (Huang and Lu, 2014). However, EtherCAT has strong requirements in terms of hardware support—specific controllers are required for 'slave' hosts—and topology—only lines and rings are allowed.

The original Ethernet has been extended to allow sharing a Network Interface Card (NIC) between multiple guest machines, where, once again, it is shown that paravirtualization offers better performance compared to emulated contexts (Motika and Weiss, 2012). (Dong et al, 2012) propose the use of the single-root I/O virtualization (SR-IOV) standard for sharing the Network Interface Card. This technique uses PCI Express (PCIe) and needs to be supported by the hardware device.

The typical approach for Ethernet virtualization consists of having a virtual machine acting as a proxy for the others. For example, the hypervisor Xen (Barham et al, 2003) uses bridges for connecting the VMs to the device, and research is ongoing on how to improve this type of communication for real-time scenarios (Li et al, 2015, 2022). An alternative technique involves having a specific VM for managing the NIC. This method is at the base of our VTSN

Proof-of-Concept, and it has also been explored by the Ethernet virtualization investigation by (Borgioli et al, 2022). Although their work considers similar reference hardware and virtualization designs, it uses the proprietary CLARE hypervisor and focuses on the memory regulation capabilities offered by *QoS-400*. Our approach instead is based on the open source Jailhouse hypervisor, targets the real-world integration and virtualization of TSN technologies. A comparison with the Ethernet virtualization approach proposed by (Borgioli et al, 2022) cannot be performed because it is based on a closed-source hypervisor. However, we compared our solution against Xen, and we show that we greatly outperform it in Section 4.3.

In recent years, the virtualization of TSN started to be addressed. (Biondi et al, 2021) advocate the use of this technology for autonomous driving architectures. (Leonardi et al, 2020; Caruso et al, 2021) propose to enable TSN communications in heterogeneous platforms handled by a hypervisor, while (Garbugli et al, 2022) propose a novel approach to support the TSN protocol in virtual machines through a precise clock synchronization method. In (Garbugli et al, 2023), a solution is presented for meeting the time-sensitive requirement in containers based on Kubernetes (Cloud Native Computing Foundation, 2014).

While the use of TSN technology has been explored in V2X communication scenarios (Ding et al, 2022; Boutin et al, 2021), Ethernet TSN is a natural match for in-vehicle communication in autonomous driving (AD) scenarios. As highlighted by (Brunner et al, 2017), ECUs in commercial vehicles are growing in number and complexity. This increases the need for a communication technology like TSN to guarantee the functional safety of the vehicle. (Lee and Park, 2019) proposed a TSN integrated environment simulator and measured the overall reduction of end-to-end latency in autonomous driving use-cases, while (Park and Park, 2023) presented the use of Time-Sensitive Networking for zone-based in-vehicle network architecture. The benefits of TSN for in-vehicle communications are also explored by (Farzaneh and Knoll, 2017), who developed an experimental test bench to demonstrate the low latency and jitter of Ethernet TSN technologies.

The aforementioned work, regarding TSN-enabled autonomous driving architectures, used emulated environments. In this work, we have deployed and evaluated the actual possibility of using a TSN-enabled autonomous driving architecture in a real environment.

3 Architecture Description

In this section, we describe the different components that are embedded in our car to form a complete TSAA solution. As shown in Figure 1, our car was modified to enable research projects like this one. It has a custom *low-level vehicle controller*, implemented in a Centralized Controller Unit (CCU), which gives full control of both the steering wheel and the throttle/brake pedals via CAN bus signals. In general, the tasks performed by an autonomous

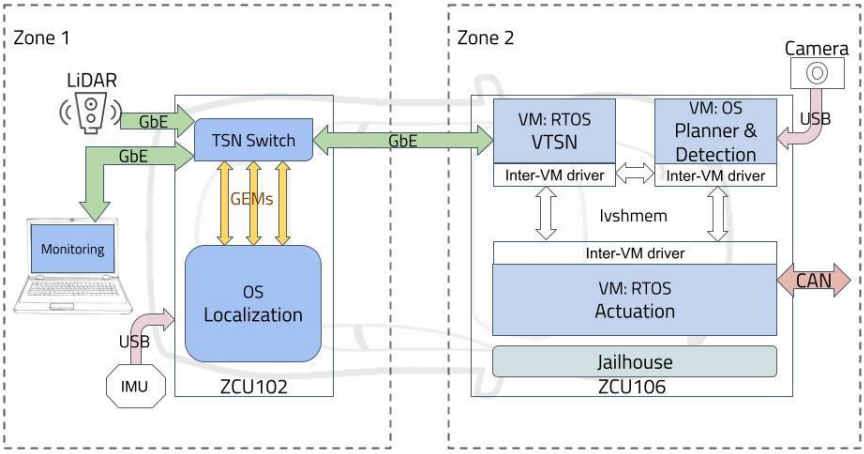


Fig. 2 Our future architecture for TSAA.

driving system are to follow a reference trajectory, to detect and avoid possible obstacles. As depicted in Figure 2, our self-driving software stack features the following key elements:

- Localization: provide the coordinates of the vehicle in its surrounding area according to the information gathered by the different sensors ;
- Detection: allow the detection of other vehicles, pedestrians, etc., to avoid collisions and a safe journey ;
- Local planning: dynamically generate alternative trajectories according to the position and detected obstacles ;
- Actuation layer: determine the correct and safe commands to send to the CCU to follow the selected path according to the current position ; and
- Monitoring: display current information about the system state.

In the near future, more and more features will be added to autonomous driving systems, which will require an ever-increasing amount of computational power (more ECUs) with more communication, real-time guarantees, and predictability, even for communications. As stated in (Fumio et al, 2022; Robert Bosch GmbH, 2017), the current auto-market is pushing for a multi-zonal architecture in place of a multi-domain one. A zone groups ECUs/sensors based on their proximity instead of their functionalities. The main benefit of having multiple zones lies in the reduction of the necessary wires to connect the involved components.

Since realizing a multi-zonal architecture with a different number of ECUs has a high cost, we are limiting our prototype to a dual-zone architecture, which is enough to test and evaluate a TSAA system. Therefore, in our use-case architecture, depicted in Figure 2, we identified two zones:

1. *Zone 1* to localize and monitor the vehicle,
2. *Zone 2* to control the vehicle (accelerating, steering, braking, . . .).

This design is still extendable by either augmenting zones or by adding more, for example, for infotainment.

Due to the heavy computational demand, in addition to zones, a typical design solution for autonomous driving systems is to partition the computation between multiple computing platforms.

The following subsections describe our choices regarding the different hardware, middleware, and software components.

3.1 Sensors

As a constraint from our industrial partners, the autonomous driving system has three sensors:

- a Velodyne VLP-16 Light Detection And Ranging (LiDAR) (Velodyne, 2022);
- an Xsens MTI-G-710 Inertial Measurement Unit (IMU) (XSens, 2022);
- a FFY-U3-16S2C-S-DL camera (Teledyne FLIR, 2019).

By pulsing infrared light, and measuring the return travel time upon colliding with target objects, the LiDAR maps the distance between itself and all objects surrounding it. The VLP-16 provides the ranging data of 1800 points for each of its 16 vertical channels on a 360-degrees field of view and delivers them via an Ethernet connection. The sensor packets are transmitted by the LiDAR at a frequency of $10Hz$.

The IMU provides an accelerometer, a magnetometer and a gyroscope, which are used to estimate the movements and rotations of the vehicle. The IMU exposes two interfaces: a parallel serial (RS232/RS422/UART) and a USB. The latter was chosen as a physical interface for ease of use and port availability on the used ECUs. The IMU provides data at a frequency of $400Hz$.

The camera generates an image stream at 60 frames per second with a resolution of 1440x1080 and is connected via USB.

All the sensors are used for the navigation of the vehicle in its surrounding environment. The combination of their gathered data allows them to provide a robust and accurate localization. In particular, the IMU provides position information at a very high frequency but with low measurements accuracy, while the LiDAR provides high accuracy measurements at a much lower sampling frequency. Moreover, the high frequency of the IMU allows it to correct the LiDAR distortion. For a similar reason, the fusion of camera and LiDAR data allows for high accuracy and responsiveness in object detection, such as pedestrians.

Note, the CCU, already present in the car, also provides speed and odometry information that can be used for safety and debugging purposes or to increase localization algorithm precision. We reserve their usage for future work.

3.2 ECUs

We target two well-known platforms from the Xilinx Zynq UltraScale+ MPSoC ZCUs family: ZCU102 and ZCU106. They both feature a multi-core CPU and an FPGA accelerator that serves as an energy-efficient co-processor(Qasaimeh et al, 2019). Such platforms are widely adopted in several application domains, e.g., automotive, autonomous drones, computer vision, etc (AMD Xilinx, 2022b). Nonetheless, FPGA-based architectures are less widely adopted than General-Purpose computing on Graphics Processing Units (GPGPUs), due to their complexity, but have shown to provide a comparable or higher Performance/Watt trade-off and an increased predictability (Brilli et al, 2018; Liu et al, 2019).

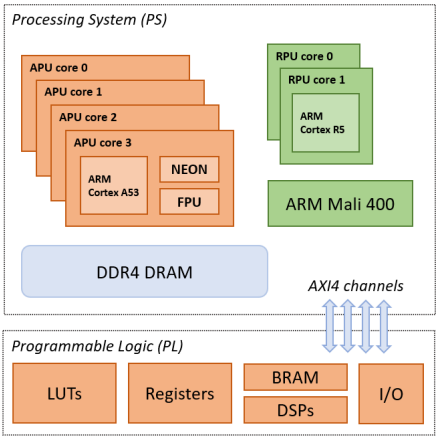


Fig. 3 Target architecture of a Zynq UltraScale+

As shown in Figure 3, the two platforms feature two CPU clusters: one with 4x ARM Cortex-A53 cores and the other with 2x ARM Cortex-R5F cores. Both include an FPGA with slightly different characteristics: the ZCU102 is composed of $600K$ system logic cells, $32Mb$ of memory, and 2520 Digital Signal Processor (DSP) slices, while the ZCU106 consists of $504K$ system logic cells, $38Mb$ of memory, 1728 DSP slices, and has a Video Codec Unit.

The FPGA of the ZCU102 is occupied by a further described TSN connectivity module, while the ZCU106's FPGA embeds an acceleration unit for the detection. This choice is motivated to maximize the occupancy of the FPGAs.

The sensors are placed as close as possible to their main consumer. This reduces communication latencies, granting better responsiveness. The LiDAR and IMU are then directly connected to the ZCU102 as their data is only needed by the localization. The camera is connected to the ZCU106 in order to facilitate the access to its main consumer, which is the detection algorithm that needs fresher image data and requires more reactivity than the localization. Hence, the image stream from the camera will be sent to the ZCU102 for

localization. A different design could connect the camera to the ZCU102, but it would increase the pressure on this board.

On the connectivity side, both platforms have only one built-in Ethernet port. Hence, to connect all the involved devices (including the LiDAR and monitor), the ZCU102 hosts the Opsero Ethernet FPGA Mezzanine Card (FMC) 4-port switch.

3.3 Multiport TSN Switch (MTSN)

To perform the packet switching in the network in a timely manner, we rely on the System-on-Chip Engineering (SoC-e) Multiport TSN switch (MTSN), it is sold as: *"SoC-e solution for any customer that requires an all-in-one solution to introduce TSN capabilities in their equipment"* (SoC-e, 2010). The MTSN switch comes in the form of an Intellectual Property (IP) core bitstream loadable on a FPGA. In sectors like the automotive industry, it offers a good level of flexibility, leading to great portability and future extensions, e.g., more zones, more platforms, more sensors. This IP core is designed for a variety of configurations from a simple 2-port end-point to a complex multiport TSN switch. It can be configured with a wide range of parameters, including the number of ports (up to 32) and the size of the queues for each port. These configurations can be changed using the Xilinx Vivado Tool (AMD Xilinx, 2022a).

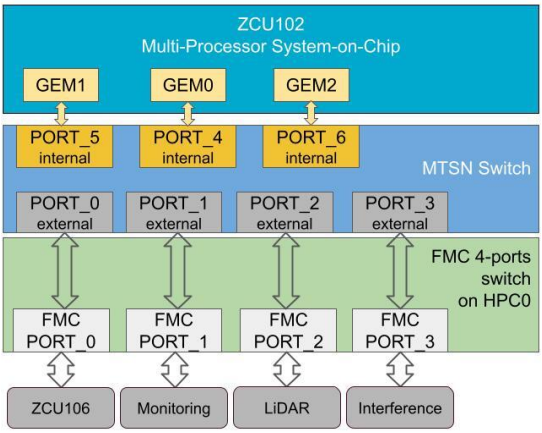


Fig. 4 Ports configuration.

Figure 4 shows our system ports configuration and how the Gigabit Ethernet Media Access Control (MAC) controllers (GEM) are connected to the *MTSN* switch to provide an interface for connecting to a $10Mbps$, $100Mbps$, or $1Gbps$ network in full-duplex capability. All four ports of the FMC located on the ZCU102 are routed to *MTSN* ports (`PORT_0,1,2,3`), in order to perform external (to the board) communication. In our setup, the ZCU102 receives

incoming camera frames through `GEM1`, and sends localization coordinates through `GEM2`. Note that `PORT_3` is connected to an external computer that we use to generate interfering traffic during our experiments, see Section 5.2. This computer would not be part of a real deployed system, and so the port would remain unused.

Of the three internal ports, `PORT_4` is used as the management port of the switch and is connected to `GEM0`, while `PORT_5` and `PORT_6` are connected to `GEM1` and `GEM2`, respectively. By describing this information in the device-tree, Linux is able to recognize how the GEMs are mapped into the port of the *MTSN* and it attaches an Ethernet interface for each of them, guaranteeing better isolation between the transited data flows.

From the various features that the *MTSN* includes, it is important to mention the full support for Virtual Local Area Network (VLAN) (IEEE 802.1Q), and the two TSN schedulers: Credit Based Shaper (CBS - IEEE 802.1Qav) and Time Aware Traffic Shaping (IEEE 802.1Qbv). The Qbv scheduler separates the communication on the Ethernet network into fixed duration, periodic time cycles, while Qav permits the reservation of the maximum bandwidth needed by traffic classes with different criticality levels or priorities. In (Zhao et al, 2022), a comparison of various individual traffic shapers was conducted across different network topologies, evaluating their end-to-end latency bounds. The experiments revealed that Qbv outperforms Qav. Despite this finding, the paper also proposes the combination of both protocols, as also suggested in (Meyer et al, 2013) and (Alderisi et al, 2012). The combination of Qav and Qbv can provide a more robust and reliable communication infrastructure for safety-critical applications. Qav can dynamically manage bandwidth allocation for different flows, while Qbv guarantees deterministic timing for critical flows, minimizing potential delays and jitter. When using both protocols together, it is essential to carefully configure and coordinate their settings to avoid conflicts and ensure optimal performance. Proper configuration of time slots and credit assignments is crucial to achieving the desired real-time behavior and bandwidth prioritization. Additionally, the specific hardware and software components in the network must support both Qav and Qbv. Compatibility between devices and adherence to the TSN standard are essential to achieving seamless integration and interoperability.

In our case-study, we only rely on Qav, as Qbv requires a global notion of time shared by all platforms, and thus requires additional mechanisms for time synchronization which are not available at the moment from the upper OS layer. Moreover, the end-to-end latency upper-bound of Qav is suitable for our current application. As of now, the combination of both protocols is reserved for future work.

SoC-e's implementation of Qav gives full control to the user, i.e., it allows to set up to 8 priority queues (one per priority level), where different priorities can share the same queue. For each couple $(queue, port)$, the user can decide to enable CBS, which schedules the different queues based on the given credit and priority values. This requires to also set the *idleSlope* value to specify

the slope of increasing credits as a percentage. If CBS is disabled for a particular $(queue, port)$, then no TSN regulation is employed on this data flow. A configuration example for a queue is given by Listing 1. We experimented with different configurations for the queue in Section 5.2 to show how these parameters can affect the transmission of the packets.

Listing 1 Example of a TSN queue configuration

```
PORT Name : PORT 0
Priority Queue : 7
CBS enable : TRUE
Idle Slope : 30
```

In our setup, we require six data flows with different priorities. From the ZCU102, both data flows coming out (localization coordinates), and coming in (camera feed) are of high criticality and require a high priority. The data reported by features for monitoring purposes are, however, of less criticality and need a lower priority. More details on the final configuration and data flows arising in the car are given in Section 5.2 when describing the experiment.

3.3.1 TSN Protocol IEEE 802.1*Qav*

This section acts as a reminder on CBS, defined in IEEE 802.1*Qav*. It schedules the traffic in sorted queues based on an Ethernet frame's priority level (7 highest to 0 lowest) given by the Priority Code Point (PCP), a 3-bit field defined in the IEEE 802.1*Q* standard (Figure 5).

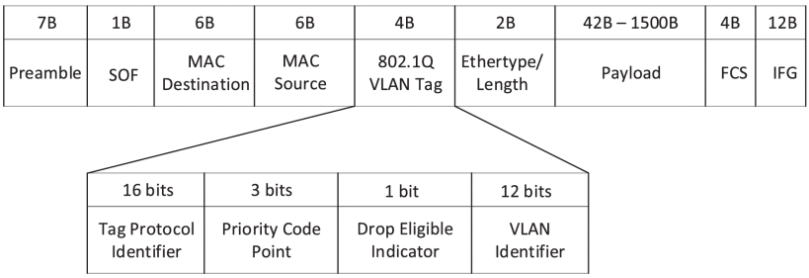


Fig. 5 (Lo Bello and Steiner, 2019) Typical Ethernet frame including VLAN Tag fields.

Each queue has a credit value used by the algorithm to determine if the queue is allowed to forward a frame through the network or not. From the definitions in (IEEE, 2018), the credit changes according to 2 parameters:

1. **idleSlope**: the charging rate of credit, in bits per second, when the value of the credit is increasing ;
2. **sendSlope**: the discharging rate of change of credit, in bits per second, when the value of the credit is decreasing.

$$sendSlope = idleSlope - PortTransmitRate \tag{1}$$

$$BandwidthFraction = \frac{idleSlope}{PortTransmitRate} \tag{2}$$

In (Mohammadpour et al, 2019), a summary of the forwarding rules for the CBS is given. The main principle is pictured in Figure 6, and briefly presented as follows:

- If the transmission line is free, the scheduler transmits a frame from the highest priority class that satisfies all of these conditions:
  1. its queue is not empty ;
  2. it has a non-negative credit.
- The credit of a traffic class is reduced linearly with the rate *sendSlope* when the class transmits.
- The credit of a traffic class increases linearly with rate *idleSlope* when the following conditions hold simultaneously for that class:
  1. its queue is not empty ; and
  2. other classes are transmitting.
- Whenever a traffic class has a positive credit and its queue becomes empty, the credit is set to zero: this is called a credit reset.
- If the credit is negative and the queue becomes empty, the credit increases with the rate *idleSlope* until the zero value is reached.

CBS permits the regulation of the maximum bandwidth that a specific traffic class can use, ensuring the stability of critical traffic over the best-effort.

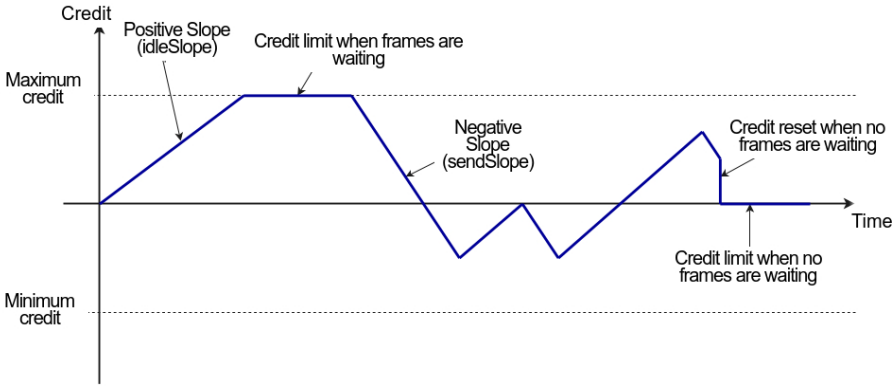


Fig. 6 Credit evolution in *Qav* from (SoC-e, 2010).

For this prototype, where multiple sources generate critical traffic, *Qav* debursts the lower critical traffic that can interfere with the critical flow, ensuring communication stability and predictability. The high level of customization provided by this protocol makes it an efficient tool for ensuring that communications data flows are secure while other traffic runs on the same network. Moreover, its simplicity allows us to select it as the first algorithm that our VTSN solution will provide.

3.4 Virtualization & OSes

Hypervisors allow to follow the aforementioned partitioning design principles. They enable multiple Operating Systems (OS) to execute on the same hardware platform without interfering with each other. Using hypervisors also allows to reduce costs and fully utilize all the capabilities of the hardware platforms while still guaranteeing temporal and spatial isolation. A hypervisor can indifferently run different OSes (e.g., Linux), Real-Time OSes (e.g., Erika), or even bare-metal applications.

Jailhouse (Jailhouse, 2015) is an open-source, Linux-based, static partitioning type-1 hypervisor whose design has been condensed to include only essential features in order to serve as an appropriate base for certifiable software. This is accomplished through a simple, yet highly effective method. The system is first booted using Linux, and then the hypervisor is loaded through a Linux Kernel Module. Moreover, static partitioning means that it does not emulate hardware resources, but rather distributes them into isolated virtual systems, known as *cells*, allowing for guaranteed resource access and predictable performance. This makes Jailhouse an excellent choice for safety-critical environments.

Jailhouse includes an *Inter-VM Shared Memory* (IVSHMEM) device that allows messages to be exchanged between virtual machines. This device is being seen by the attached virtual machines as a virtualized Peripheral Component Interconnect (PCI) device with different shared memory regions that can be used to communicate with other peers. Jailhouse defines the IVSHMEM device with some useful features, including the ability to connect up to 65536 peers, different permissions on the shared memory regions, support for life-cycle management, event signaling via interrupts, and much more. The IVSHMEM device has a register region with five 32-bit registers, each with a specific role; e.g., the *Doorbell register* is a write-only register used for triggering an interrupt into a targeted peer. Table 1 presents the usually used standard permissions of the IVSHMEM memory regions over `N` attached peers.

Table 1 IVSHMEM shared memory region permissions, where R stands for read and W for write.

Memory Region Name	VM 0	VM 1	...	VM n
State Table[1]	R	R	R	R
R/W Section[2]	R/W	R/W	R/W	R/W
Output Section VM 0	R/W	R	R	R
...				
Output Section VM n	R	R	R	R/W

[1]Memory region for saving the state of the peers.

[2]Memory region with R/W permissions for all the peers.

We performed a preliminary feasibility study of this mechanism in order to check if it could support the necessary 1*Gbps* Ethernet connectivity. We

considered Linux-based applications as well as bare-metal ones. It showed us that the IVSHMEM device can sustain such speed in both environments.

In our TSAA design, we employ two different OSes: Linux and Erika Enterprise v3. While Linux needs no more introduction, Erika Enterprise v3 is an open-source AUTOSAR RTOS for automotive environments. It supports different architectures, such as Aurix, ARM Cortex-M, Cortex-R, and much more. Its kernel offers the essentials for setting up a multi-threading environment. It implements stack sharing techniques, semaphores, three alternative scheduling algorithms, and the OSEK Implementation Language (OIL), which is an OSEK standard for statically defining the setup of real-time applications. The main issue with Erika is its lack of Ethernet driver, we therefore need a solution through virtualization to connect it to the network. With the help of Jailhouse and its IVSHMEM device, it is possible to overcome this issue and allow the actuator running on Erika to receive the localization information.

Having such two OSes allows for performance as Linux is able to benefit from the different hardware accelerators present on the platform, i.e., the FPGA for the detection. However, Linux does not offer guarantees regarding its responsiveness. Hence, for critical cases such as actuating the driving commands, the certified Erika comes in handy and increases the safety of the whole system as well as reducing its certification time with enhanced predictability.

We therefore have Linux running alone on the ZCU102 to support the heavy computation of the sensor fusion and localization. On the ZCU106, Jailhouse is configured to embed a Linux OS for the detection, a Erika RTOS for the actuation. In consequence, multiple VMs on the ZCU106 need to access the TSN-enabled network. In (Leonardi et al, 2020), a key design challenge is to understand where the TSN scheduling algorithm should be executed to maintain high performance without affecting the complexity of the hypervisors. Section 4 paves the way for a possible solution that locates the TSN scheduling algorithm into an additional VM which executes a bare-metal application to manage the single Ethernet port. Using a specific proxy VM also allows the Erika RTOS to receive/send data through the network as this RTOS is not shipped with an Ethernet driver. Moreover, this solution reduces costs and energy consumption since it does not require an additional Ethernet FMC card.

3.5 Middleware Communication

The different features of the self-driving application are implemented on top of **Robot Operating System 2** (ROS2) (Macenski et al, 2022), a distributed robotic development framework. ROS2 offers a runtime environment based on a publish/subscribe middleware referred to as Data Distribution Service (DDS). The DDS implementation chosen for this scenario is FastDDS, a free and open-source middleware developed by eProsima (eProsima, 2019). FastDDS is compliant with the Object Management Group (OMG) Real-Time Publish-Subscribe (RTPS) 2.2 and OMG DDS 1.4, thus providing publisher-subscriber communications over the UDP/IP protocol stack.

Since a robotic system code base is inherently complex, ROS2 enhances software modularity by modeling each software element as a *node*. Each node implements one or more specific functionalities, e.g., reading data from a sensor, building a map of the surrounding environment, etc. Since nodes are expected to exchange data between them, ROS2 also defines the abstraction of communication channels with *topics*. A topic is identified by a string and a message type. Each node declares a set of *publishers* and *subscribers*. Each publisher allows the node to send messages on specific topics, while a subscriber receives the messages sent.

ROS2 provides Quality of Service (QoS) policies, allowing to tune communication between nodes. These policies can define different aspects of a communication, such as the reliability or durability of messages. The ROS2 software stack is highly scalable, especially with respect to the communication middleware; hence, it is possible to implement custom policies for network traffic management/shaping directly at the middleware/user-space level. By combining a set of QoS policies, it is possible to define a QoS *profile* that can be applied to specific topics. Reliability between nodes means that a publisher of a topic will not send the next message if all the registered subscribers do not acknowledge the reception of the previous one.

To fully benefit from the scalability of ROS2 and to account for the constraints of our system, each aforementioned feature of the autonomous driving application is a specific ROS2 node. Similarly, each sensor generates a specific type of data, and so, their transmission is mapped to a specific topic. Moreover, additional topics are created to allow the localization node, as well as the planning node, to communicate with the actuation node.

3.6 Self-Driving Software

As presented above, the self-driving software stack is composed of five main ROS2 nodes: Localization, Detection, Local planning, Actuation and Monitoring.

To localize the vehicle, the localization uses the three sensors, i.e., LiDAR, IMU and camera, for an optimum compromise between data efficiency and accuracy. However, to get the highly precise current localization of the AV, a sensor-fusion algorithm needs to merge the information from the different sensors, hence exploiting the best of both of them. To perform the sensor-fusion, we use the state-of-the-art GPS-free precise localization algorithm Fast and Tightly-Coupled Sparse-Direct LiDAR-Inertial-Visual Odometry (FAST-LIVO) (Zheng et al, 2022). It is an evolution of FAST-LIO (Xu and Zhang, 2021) which appends camera data to estimate the position of the vehicle. We allocate the whole ZCU102 to the localization due to the high computational power required by the algorithm to fuse sensor data and estimate the position of the vehicle. The estimated coordinates can then be sent to the actuation for maneuver computation.

The detection algorithm receives the information from the camera. Each image is processed by a neural network based on YOLO v3 which is accelerated

by a Deep-Learning Processor Unit (DPU) core instantiated on the FPGA of the ZCU106. Recall that we cannot have the detection algorithm running on the ZCU102 as its FPGA is mostly utilized for the MTSN switch.

The detection and actuation are isolated and instantiated into two different VMs on the ZCU106. This isolation is performed through the virtualization provided by Jailhouse. Using the IVSHMEM feature of Jailhouse, both VMs can communicate. Due to its high-criticality properties (failing to control the vehicle can have dramatic consequences), the actuation task is encapsulated into a single-core VM and ported to Erika Enterprise v3 (Evidence, 2017), enhancing safety and predictability.

4 Virtual TSN: a Proof of Concept

The VTSN solution aims to enhance TSN features for multiple VMs running on the same platform, on which the available Ethernet ports are fewer than the number of VMs. In other words, VTSN improves the predictability of data flow for multiple VMs accessing the network through a single virtually shared Ethernet port. The difficulty of such technology lies in the necessity to properly schedule the generated Ethernet frames transmitted to/from the various VMs.

To implement a VTSN solution in a system, the choice of the hypervisor plays a crucial role in achieving the desired level of predictability and security. Albeit full virtualization hypervisors, such as Xen, can be employed, our Proof-of-Concept is restricted to static partitioning hypervisors, such as Jailhouse. Static partitioning hypervisors have a smaller code base and fewer layers of abstraction, resulting in less variability, less overhead, and more deterministic behavior in terms of timing and performance. With this approach, the Ethernet port access is assigned exclusively to a single virtual machine, allowing it to handle and control network traffic without any interference from other VMs.

Our VTSN solution is based on the Inter-VM shared memory (IVSHMEM) feature, which needs to be provided by the hypervisor, as introduced in Section 3 for Jailhouse. Using IVSHMEM helps reducing the amount of copy operations necessary for exchanging messages between VMs. Moreover, this communication must also provide a wake-up mechanism, e.g., by sending an interrupt to the interested VM.

This solution is then composed by two main components:

1. a virtual Ethernet switch; and
2. at least, TSN protocol.

In Section 5, we will demonstrate the effectiveness of TSN on a complex system performing in a real and simulated environment. Hence, this Proof-of-Concept focuses solely on the virtual Ethernet switching mechanism, which when combined with TSN will form a complete VTSN solution. At the time of writing, the last software brick that we are missing is a traffic scheduler that will enforce the *Qav* or *Qbv* protocols, which we argue to be important but less technically challenging as the code base exists.

4.1 Virtual Switch

Figure 7 illustrates an overview of a possible system enabling VTSN. The central point of our solution is a specialized VM, hereafter named the Virtual Switch (VS), which allows other VMs to send and receive Ethernet frames to and from the TSN network connected to it. The VS is the only VM having access to the Ethernet port; therefore, any VM's traffic needs to pass through it, enabling a single scheduling point. For brevity, we hereafter use VMs to refer to all VMs, excluding the VS if not specifically stated.

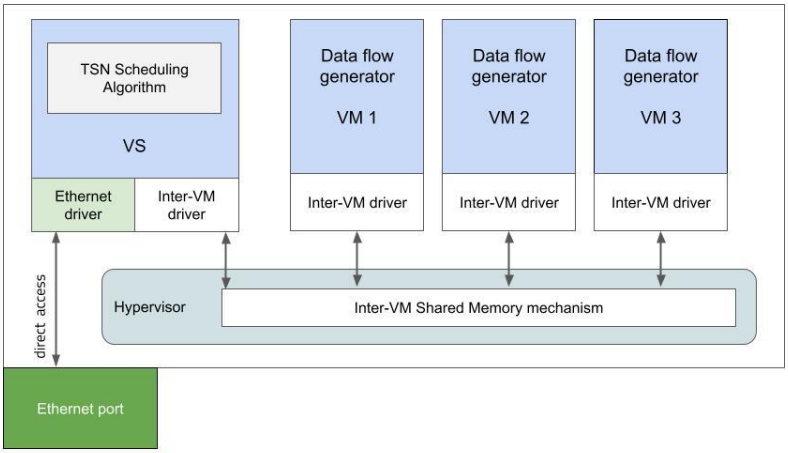


Fig. 7 The representation of a system with VTSN and multiple virtual machines running an application and generating data.

To enable communication between the different VMs and the VS, they are all connected to a dedicated shared memory using the IVSHMEM device. This shared memory region is divided into sections, on which different permissions are applied in order to ensure safety and security through address space isolation. As a result, a section cannot be assigned to multiple VMs, and the number of sections assigned to a VM is only limited by the memory size and other VMs requirements. This means that the Ethernet frames sent by, e.g., VM0 cannot be read or overwritten by, e.g., VM1. Figure 8 pictures the mapping layout of these shared memory regions. For each VM, we assigned a section with R/W permissions, shared with the VS. Moreover, the section itself is divided into three zones in order to exchange Ethernet frames. In the first one, we are saving the metadata of the circular buffers that handle the other two zones. The second one is used by the VS to write the receiving external Ethernet frames to the VM, while the last one is used by the VM to write the Ethernet frames that need to be sent. Obviously, VS reads from the VM's zone and vice versa.

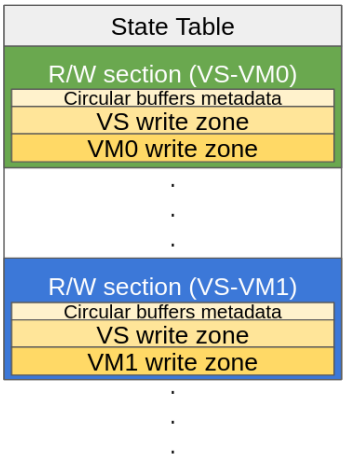


Fig. 8 A representation of the IVSHMEM mapping layout.

We implemented VS as a bare-metal application using *Newlib* (RedHat, 1999), a C standard library designed for embedded systems. Due to its traits and the communication mechanism that it provides, Jailhouse has been selected for hosting this version in a single-core VM. For implementing VS, bare-metal programming makes it easier to develop small drivers and prove the feasibility of an idea. It does not have multitasking ability, nor the presence of a scheduler, hence allowing further timing analysis and facilitating certifications.

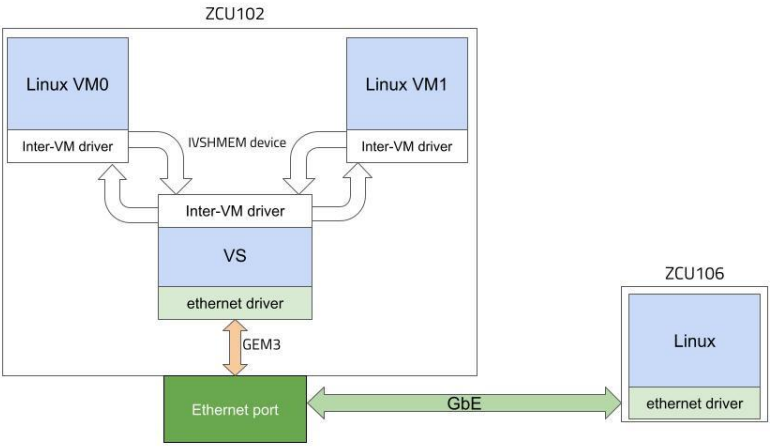


Fig. 9 A representation of the experiments setup.

On the VM side, we develop an Inter-VM driver in Linux (frontend). The driver is an adapted version of the `ivshmem-net` driver, from (Jailhouse, 2020),

that uses the circular buffer and IVSHMEM device to communicate with the VS (backend).

4.2 Evaluations

In order to evaluate our driver's performance, we experimented the following four configurations on the ZCU102 board:

1. A Linux VM with direct access to a GEM (*Conf.1*) ;
2. VS and a Linux VM (*Conf.2*) ;
3. VS and two Linux VMs (*Conf.3*);
4. VS and three Linux VMs (*Conf.4*).

This first system is directly wired to a ZCU106 board, as shown in Figure 9. The Linux systems are hosted by Jailhouse and built with the PetaLinux Tools (AMD Xilinx, 2020), a tool that provides a set of functionalities useful to build, develop, test, and deploy an embedded environment, including a Linux distribution. Configurations 2, 3, and 4 have also been compared to a similar one where the Xen hypervisor is employed instead of Jailhouse. The setup is detailed in Section 4.2.1.

Note that three Linux VMs are the maximum number of VMs that we can run on the ZCU102 using only the Cortex-A cluster (4 cores).

Software components versions used are summarized below.

- The Linux kernel version is the 5.15.0-xilinx-v2022.2 ;
- The ping tool is provided by BusyBox version 1.34.1 ;
- The iperf2 version is 2.0.13 ;
- The Jailhouse version is 0.12 ; and
- The Xen version is 4.13.0.

To measure networking latency, we use the `ping` tool, from (iputils, 2022), to trigger and measure ICMP packets round trips between the ZCU102 and the ZCU106. In configurations 1 and 2, the Linux VM is executing ping, while in configuration 3 and 4, both Linux VMs are pinging each other simultaneously. Experiments were executed using different packet sizes—from 64 bytes up to 1024 bytes, using the powers of 2, including also 1480 bytes, i.e. the maximum size allowed by the ICMP packets. Each experiment transmits $100,000$ packets.

We also measured the maximum reachable bandwidth using `iperf2` (Dugan et al, 2016) tests on the same previous configurations. This tool allows us to measure the maximum TCP and UDP bandwidth as well as other characteristics. `iperf2` usually requires at least two entities: a server and a client. Once the server is active, the client can start sending Ethernet frames to the server. Both entities measure the bandwidth reached. In this case, we installed the server on the ZCU106 and tested the VMs as clients. Bandwidth experiments were executed for $30s$ (`-t` option) for both TCP and UDP protocols. The bandwidth set (`-b <bandwidth_value>` option) and used for the UDP tests is equal to $1Gbps$.

4.2.1 Xen Configuration

Xen (The Linux Foundation, 2003) is a bare-metal hypervisor that allows the creation and management of multiple virtual machines on a single machine. This hypervisor refers to virtual machines as domains. The first virtual machine to boot has privileges, referred to as *Domain0*(Dom0), and can manage the unprivileged domains, known as *DomU*s. We evaluated this hypervisor Ethernet virtualization features using a paravirtualized configuration that reflects, as much as possible, our VS. Hence, Dom0 acts as VS by granting Ethernet access to the three Linux VMs, *DomU1*, *DomU2* and *DomU3*. To properly implement a static partitioning setup, like the Jailhouse's ones, we used the `null` scheduler, which pins each virtual CPU to a physical CPU, and we assigned one core for each virtual machine.

In order to provide Ethernet connection to the *DomU*s, we followed the most configuration suggested by the Xen Networking Guide (Xen Project, 2018)—a software bridge in *Dom0* connecting the *DomU*s. As a consequence, in order to have Ethernet access, virtual machines are attached to this bridge by declaring a virtual interface (vif) in their configuration files. We also tuned the networking configurations according to the Xen Performance Networking Guide (Xen Project, 2014), adopting, inter *alia*, the recommended TCP settings for *Dom0*, *DomU1*, *DomU2* and *DomU3*.

4.3 Results

Experiments results are cleaned up by the worst 0.002% measurements, in order to remove outliers caused by uncontrollable external factors such as hardware failures or kernel subroutines execution. We then compared the last three configurations with a Xen-based solution.

Figure 10 shows the min, average, and max latency of each configuration. For the last two configurations, we reported the measured latency per VM. In this experiment, we can see that the introduction of the VS in the system brings a slight increase in latency, but at the same time, it reduces the standard deviation, thereby improving the stability. Moreover, VS allows the communication with the network with two VMs slightly impacting the cases with small packet sizes, but remaining quite the same for larger ones. By comparing a Xen-based solution with our VS, it is clear that with VS the latencies are lower in all the cases, as shown in Figure 11. Moreover, the worst case for our solution is still better than the best case for Xen.

Bandwidth test results obtained with iperf2 tool are shown in Figure 12. They emphasize that our solution is not currently able to reach the maximum possible bandwidth for both protocols; however, it is better than Xen. Moreover, due to the actual round-robin schedule, the bandwidth is split correctly between the two or three running VMs.

On the receiver side, VS is able to reach the theoretical Gigabit bandwidth for both protocols and fairly split the bandwidth between the different running VMs. Xen is always outperformed by VS, especially with the UDP protocol,

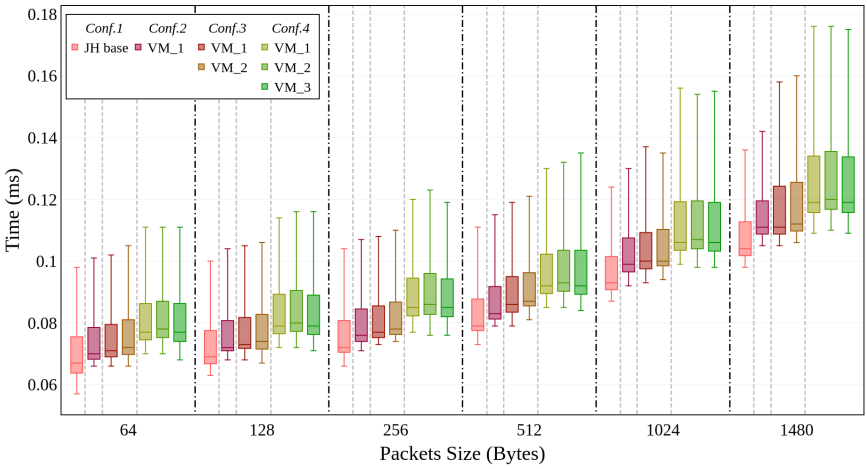


Fig. 10 Ping results over 100,000 packets using an interval of 1 ms between two consecutive ICMP messages (ping `-i` option). The configurations referenced as *Conf.X* correspond to the configurations listed in Section 4.2.

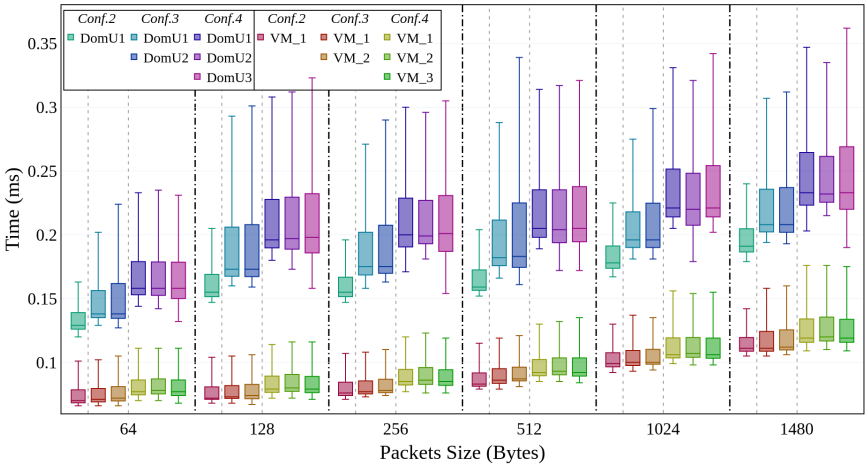


Fig. 11 Ping results over 100,000 packets using an interval of 1 ms between consecutive ICMP messages (ping `-i` option). The figure compares the Xen-based solution (DomUX) and our *VS*. The configurations referenced as *Conf.X* correspond to the configurations listed in Section 4.2.

where to avoid a drastic drop in performance, we had to limit the bandwidth of the iperf client with the `-b` option.

In conclusion, the first evaluation of the VS prototype is quite satisfactory. However, still missing TSN traffic control features, as well as detection stack from the self-driving vehicle, VS was not considered for inclusion in our real use-case scenario.

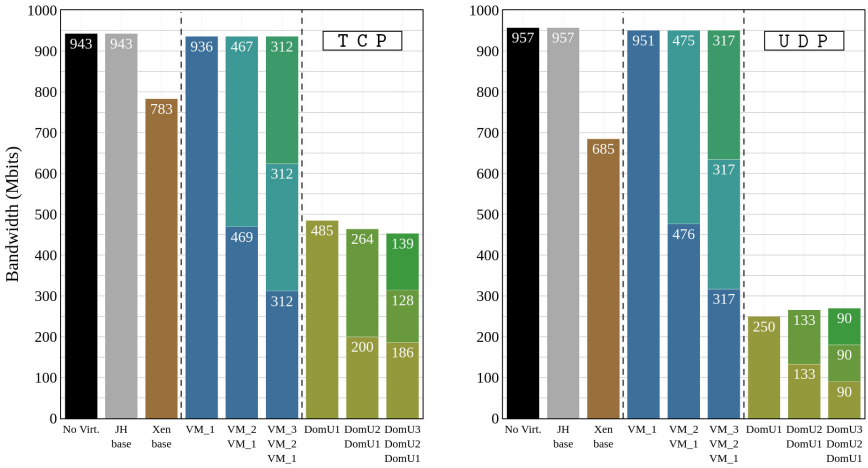


Fig. 12 Bandwidth measurements using `iperf` as client for both TCP and UDP protocols. The figure illustrates the achieved bandwidth in various configurations: a non-virtualized environment, virtual machines with direct GEM access on respective hypervisors (*Conf.1*), and two solutions utilizing *VS* as the backend and *Dom0* in a Xen-based setup (*Conf.2* to *Conf.4*).

5 Deploying TSAA on the Road

We selected, adapted and integrated a realistic autonomous driving use-case into the first iteration of Time-Sensitive Autonomous Architecture. Before proceeding with the architecture design for our use-case scenario, we provide a more detailed description of the autonomous driving system that has been deployed and tested.

5.1 Autonomous Driving System

The purpose of this autonomous driving system is twofold. In the first place, while the car is being navigated manually by a person, the system has the objective to record live sensor data. The sensor data is then used by the offline mapping software to produce a high resolution point cloud map of the environment. We define this as the mapping phase. After the mapping phase is complete, the objective of the autonomous driving system is to steer the vehicle along a given reference path. We define this as the autonomous driving phase. To achieve these objectives, multiple ROS2 nodes are implemented.

During both phases, the sensor driver nodes collect sensor data from the IMU and LiDAR to make it available inside the ROS2 middleware. In the mapping phase, the sensor data is recorded as a ROS2 bag file that will be used for offline mapping, while in the autonomous driving phase the previously generated map is used by the localization node in conjunction with the sensor data to localize the vehicle in the mapped environment. The localization output is a ROS2 *Odometry* message with an estimate of the vehicle pose composed of a three-dimensional position and a quaternion describing the rotation state

of the vehicle. This data is then passed to the high-level actuation node that keeps track of the vehicle's position and generates an adequate speed and steering value to maintain the vehicle on the desired reference path. Finally, the drive parameters are passed to the low-level actuation layer, whose job is the smoothing of the vehicle commands and their forwarding via the vehicle's CAN network to the CCU, which in turn controls the engine and steering wheel. Here follows a more detailed description of the software solutions implemented for each phase.

The first sensing layer contains the LiDAR and IMU drivers, which transform the raw data of the sensor into ROS2 messages. LiDAR messages are published at $10Hz$ rate, using an average bandwidth of $5.0MB/s$. The IMU messages, on the other hand, are published at a frequency of $400Hz$, with an average bandwidth of $160KB/s$.

In the literature, various methods have been proposed to solve the Simultaneous Localization And Mapping (SLAM) problem, like LIO-SAM (Shan et al, 2020) and FAST-LIO (Xu et al, 2022). While FAST-LIO provides a better SLAM latency, it comes at a cost: FAST-LIO does not implement any type of loop closure, so it can accumulate some long-term odometry drifts and deviations in the mapped environment. On the other hand, LIO-SAM is computationally heavier, as it implements feature extraction and loop closure methods. The feature extraction phase generally reduces noise inside the generated map, while a loop closure method helps to correct drifting issues. This makes LIO-SAM optimal for the mapping phase, as the maps generated with it depict a more realistic environment when compared against FAST-LIO generated maps.

The localization is the most complex and critical node in the autonomous driving phase since any communication delay from and to this node can lead to a higher reaction time of the actuation software. The main objective of this node is to filter the position and orientation of the vehicle in the environment. The vehicle pose is continuously estimated using the live sensor data and published to a ROS2 topic. Given the embedded nature of the project and the low computational power of the ECUs used, FAST-LIO was chosen as it provides a better localization latency.

The goal of FAST-LIO in the localization layer is to provide an accurate estimate of the vehicle odometry inside the mapped environment. To reduce the localization latency, FAST-LIO leverages iKD-trees (Cai et al, 2021), a data structure used to index map points in an efficient way. The iKD-tree structure reduces the k-nearest-neighbor search time, which is the most time-consuming section of the software, thus improving the overall local registration latency. Since the IMU data come in at a much higher frequency than the LiDAR, the two measurements are integrated in an iterated Extended Kalman Filter (EKF)(He et al, 2023). The IMU data is also used in a preprocessing section to deskew the LiDAR point cloud, which improves localization accuracy. After the LIDAR measurement is integrated inside the EKF, its output state is the best estimate for the vehicle movement, so it is published as an *Odometry* message.

To avoid publishing high covariance poses, the system does not publish in between LiDAR measurements processing, so the estimated odometry will be published at the same LiDAR frequency ($10Hz$). These odometry messages published by FAST-LIO use an average bandwidth of $7.30KB/s$.

After the pose of the vehicle is estimated by the localization node, it is received by the high level actuation node, which determines the command to send to the CCU, through the CAN bus, in order to maintain the predefined trajectory. As described in Section 3, the actuation node has high-criticality and has been ported into a single-core virtual machine running the RTOS Erika Enterprise v3 accentuating the isolation properties gained by using a hypervisor. However, Erika Enterprise v3 does not support ROS2, hence, in order to receive the *Odometry* messages, we opted to implement a node running in another VM. This node, hereafter referred to as *ProxyROS2*, is responsible for extracting the coordinates of the vehicle from the received *Odometry* messages and forwarding them to Erika Enterprise V3 using the Inter-VM Shared Memory communication. Since we also want to monitor the actuation data, *ProxyROS2* creates ROS2 messages with the actuation data, received using the same mechanism, and sends them to *adx_gui*. This led us to choose the ZCU106 for the global planner, too. The virtualized environment is then divided into two virtual machines:

1. Erika on one core, as previously described; and
2. the global planner and the *ProxyROS2* on three cores running a Linux-based distribution.

The actuation node is composed of three parts:

1. the control algorithm;
2. a PID controller; and
3. a CAN driver.

The control algorithm used to calculate the driving parameters is based on the regulated pure pursuit approach (Coulter, 1992). In order to compute the necessary driving parameters, the node needs to be initialized with some vehicle-specific parameters like the wheelbase (the distance between the front and rear axles) and the maximum steering angle of the wheels. The pure pursuit node is also fed with a set of ordered waypoints that represent the reference path to follow. Each waypoint is composed of a three-dimensional position and a linear speed that should be kept around that waypoint. These waypoints are used at initialization time to calculate an interpolation function which will be used during the execution to interpolate the required points in the reference path.

Each time the pure pursuit node receives a pose, it geometrically calculates the steering angle required to keep the vehicle on the reference path. This is done by finding a point a certain number of meters ahead in the reference path (the look-ahead distance) and by geometrically calculating the optimal steering angle to reach that point. This look-ahead distance is adjusted based on the current vehicle speed. The output steering value is then combined with

the desired speed contained in the reference path at the current position to build the drive parameters.

Once the drive parameters (desired speed and steering angle) are computed, a simple PID controller is used to smooth the driving parameters. This step is done to avoid abrupt changes in the speed of the vehicle. The PID receives the vehicle odometry data from the CCU via the CAN network. This odometry is computed by the CCU by using the live data from the engine and steering wheel. This odometry is used by the PID in combination with the drive parameters received from the pure pursuit to compute a new set of smooth drive parameters. These are relayed to the CCU via the CAN network, which finally actuates the command by controlling the engine speed and steering angle of the vehicle.

In order to monitor the status of the nodes during execution and for safety reasons, two *visualization* nodes are presented. To monitor the general status of the mapping and localization, we deployed *RViz*, the standard ROS2 GUI. To plot and monitor the driving parameters and real-time actuation data, a custom GUI named *adx_gui* was implemented. This GUI is also used as a safety system to manually enable or disable the CCU in case of any issue with the navigation software stack. The two GUIs are meant to be used together to monitor the status of the vehicle and act as a general switchboard for the entire system.

With this autonomous driving system, we want to demonstrate the importance of TSN-enabled communications in real-time applications using TSAA-involved technologies.

5.2 Iteration Description

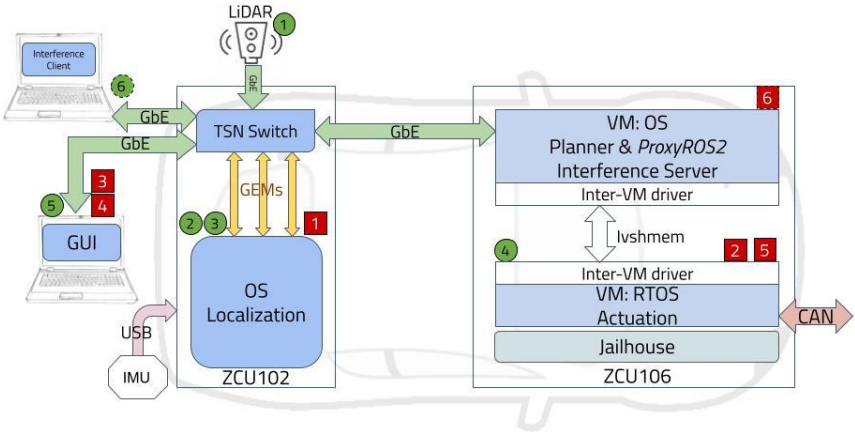


Fig. 13 System design and its data flows. The green circles represent the flows starting point, while the red squares their conclusions.

Figure 13 shows how the components and nodes were placed in this first iteration and the involved data flows. This iteration is composed of different communication mechanisms. The ZCU102 receives LiDAR sensor data over a gigabit Ethernet cable, while the IMU sensor data is collected over a USB connection. Moreover, the communication with the TSN switch is managed by different GEMs, granting better isolation between flows. The MTSN setup used for this use-case is shown in Figure 4. While on the ZCU106, there is an Inter-VM Shared Memory communication to exchange messages between the two virtual machines and a CAN bus that communicates with the vehicle CCU. Furthermore, in this iteration, we used only the A53-cluster on both boards.

The data flows involved in this system are shown in Figure 13, where numbers match between the figure and the following enumeration (green and red numbers are respectively for the start and end points of each data flow):

1. The LiDAR produces data and sends it via an Ethernet cable connected to one of the FMC switch ports. Due to hardware limitations, this sensor sends broadcast data ;
2. Once the LiDAR and IMU's data are received, the localization node uses them to calculate the car's pose. The pose is then published to the topic shared with the *ProxyROS2* node. The latter simply forwards the pose to the actuation node using the IVSHMEM device and a cache-aligned circular buffer. In this case, due to the low frequency required for this communication and its high criticality, the shared topic between the localization and the *ProxyROS2* uses the default Quality of Service which ensures reliability ;
3. The same calculated pose described in flow 2 is sent to *RViz* for the visualization. In this case, since the monitoring system is not critical, we use a different topic and QoS, which is not reliable. This ensures that the localization node will not stay in a waiting state for missing acknowledgements from this flow ;
4. Starting from the car CCU, the actuation data are being propagated in the system until they reach *adx_gui* node, where they will be displayed ;
5. As already explained, *adx_gui* is also capable of enabling or disabling the CCU. As a result, this flow has a high criticality but is activated only in emergency ; and
6. To demonstrate the benefit of TSN, we introduce this interference flow. Without TSN, this flow will introduce an ever-growing delay to messages received by the actuation. Since the localization node uses a shared topic with the actuation node and uses the same connection, the introduction of this interference flow creates noises on the Ethernet cable between the ZCU102 and ZCU106.

The complete flow number 2 (from the localization to the actuation) is the most critical of the system because any interruption or even too large delays can leave the autonomous vehicle unable to localize itself in the environment, leading to unexpected behaviors or dramatic consequences.

Due to their criticality level, *Qav* parameters, i.e., priority and *idleSlope*, must be set so that they will privilege flow 2 and limit flow 6. Hence, flow 2 must have a higher priority and bandwidth than flow 6.

The advised reader may note that in flow 2, the *ProxyROS2* node has to forward the pose to the actuation node located on the RTOS. This indirection necessitates an extra overhead. To avoid such a design, it is necessary to first have a TSN driver available for the second VM OS (here, Erika RTOS) and, second, to allow multiple VMs to access a single Ethernet port. As Erika does not provide such a driver and Jailhouse does not allow this configuration, a more evolved solution would be to virtualize the Ethernet traffic, which summarizes our first contribution in Section 4. Furthermore, we could use the VTSN to prioritize the acknowledgement flows.

Another aspect that needs to be considered is the interference flow. We have selected an additional laptop connected to the ZCU102 to generate the flow 6 in order to minimize the impact on the system.

For safety reasons, this system is previously tested in a Hardware-In-The-Loop (HiL) setup using an open-source simulator for autonomous driving named CAR Learning to Act (CARLA) (Dosovitskiy et al, 2017). CARLA is based on Unreal Engine 4, a well-known 3D graphics engine. This simulator provides multiple features needed to test autonomous driving systems.

In the first place, CARLA offers many open assets, like vehicle models and ready-made urban layouts. Secondly, CARLA supports the implementation of a CAN bus interface simulator; therefore, we developed a new CAN bus interface that simulates the CCU of the real vehicle. Finally, using the `carla-ros-bridge` package, the simulator can be directly interfaced to the ROS2 framework, on which our autonomous driving stack already runs.

We have tried our best to simulate the real environment as much as possible, nonetheless, we want to advise the reader that, albeit CARLA simulator is a comprehensive tool for simulating the real-world scenarios, it can generate a partial point cloud, and it is not currently able to correctly simulate all the real components, e.g., the generated LiDAR messages do not include all the data fields, as documented in this issue from the source repository of the previous cited package (ros-bridge Community, 2020). Keeping this in mind, the difference between the software stack on the vehicle and the one on the simulator relies on the LiDAR and IMU sensors and driver nodes, which are not used since the sensor data is directly published by the simulator bridge as ROS2 topics. Moreover, the IMU and LiDAR send data via the same Ethernet cable and work at a frequency of $33Hz$ and $14Hz$, respectively. These frequencies allowed us to prioritize the generation of a complete LiDAR point cloud.

The purpose of the Hardware-In-The-Loop setup is twofold:

1. It improves safety in the early testing phases since bugs in the software can be found and replicated without a real vehicle ;
2. It enables the functionality and performance testing of the system without the long setup times of the sensors and ECUs on the vehicle ; and
3. It reduces the costs and speeds up the researcher.

Some of the CARLA experiments are presented in Section 5.3. Figure 19 depicts the simulation environment data flows (without considering the VS solution).

Only when the system reaches a certain level of stability is it possible to proceed with the real context, where all the necessary equipment is mounted in the *Maserati Quattroporte*. Both ECUs are placed in the trunk and powered by the car's power supply system. Moreover, in order to control the system or launch/stop ROS2 nodes, two UART cables (one for the ZCU102 and one for the ZCU106) are passed through the seats and connected to the laptops (to facilitate the control of them, one for each board), so that it will be possible to use serial communications to address the command prompts. We also configure the IP addresses and VLANs on the ZCU102, ZCU106, and the two laptops, in order to use the TSN properties on the aforementioned flows.

The first testing phase requires checking that everything is properly mounted using preliminary tests such as sensor acknowledgement and connection checks. Once the preliminary checks are completed, the mapping phase starts. In this phase, the vehicle is driven manually around the test environment. This phase has two goals: one is to create a map of the surrounding environment, as described in 5.1. Secondly, the reference trajectory is recorded as a set of ordered waypoints, which will be followed later on in the autonomous driving phase. Once the mapping phase is successfully completed the car is back at the starting point, and the reference trajectory gets saved and sent to the monitoring software.

If the mapping phase is successfully completed the autonomous driving phase can start. Before enabling TSN properties and the interference component, a first round (five laps of trajectory) is completed. This first autonomous run serves as the baseline and is hereafter referred to as Scenario A. We then activate the interference flow without activating TSN, and perform a second run, referred to as Scenario B. For the subsequent runs, we keep the interference flow activated and enable TSN, allowing us to sharpen the actual effect of the activation of TSN.

At this point, an iterative experiment phase starts with the following actions while varying the TSN parameters:

1. Manually drive the car to the starting point ;
2. Enable autonomous driving system ROS2 nodes ;
3. Load the path and start the Jailhouse cell containing Erika ;
4. Configure the *Qav* parameters, priority and *idleSlope* for both VLANs ;
5. Run `iperf` as server on the ZCU106's Linux (2nd and subsequent runs);
6. Use *adx_gui* to monitor the actuation and CAN communication ;
7. Run `iperf` as client on the interference laptop's Linux ;
8. Let the car drive, meanwhile:
   (a) Wait for the completion of the five laps ;
   OR
   (b) Interrupt if the localization has been lost or the car is about to crash due to the latency in the communication of the odometry ;
9. Restart from 1.

The previous phases and steps are equal in the simulation environment, too.

The experiments were considered successful if at least five laps were completed, a failure otherwise. We evaluated different scenarios in the simulator environment; however, since experimenting in the real vehicle is expensive (at least 3 people are required), in that case, we have selected and evaluated five meaningful scenarios. Consequently, we will show these five scenarios for both environments.

Table 2 Tested scenarios and their *Qav* parameter values and corresponding results in both simulated and real environment
LF: Localization Flow, IF: Interference Flow.
✓: success run with 5 complete laps.
χ: the car was not able to complete all the 5 laps.
[1]Qav not enabled.

	Scenario									
	A^1		B^1		C		D		E	
Qav Parameters	LF	–	LF	IF	LF	IF	LF	IF	LF	IF
Priority	–	–	–	–	7	1	7	1	1	7
idleSlope	–	–	–	–	100	10	100	50	1	100
Results	✓		χ		✓		✓		✓	

Table 2 shows some of the *Qav* parameters values that were used for the experiments; these values were conveniently selected in order to evaluate the impact on the system. Moreover, the last line of the table reports the result of the corresponding test case. The scenarios A and B are executed without the usage of the TSN protocol *Qav*. The scenarios C and D privilege the localization flow by assigning a high priority and high *idleSlope* compared to the interference flow, while the scenario E privileges the interference flow instead of the localization one.

By cross-checking parameters and results, it is clear that when the localization and interference flows are running uncontrolled, the autonomous driving system is not able to achieve the goal of 5 laps, although with Qav, it is able to localize even in the scenario E, where we assign the lowest *idleSlope* and priority values to the localization flow and the highest *idleSlope* and priority values for the interference flow. As documented in the standard definition of this protocol by (The Institute of Electrical and Electronics Engineers (IEEE), 2010), the transmission of the highest priority flow can be delayed by the maximum Ethernet frame size supported by the MAC, and, thanks to the low bandwidth necessary to exchange the *Odometry* messages, this allows the vehicle to locate under such circumstances.

Next sections present and discuss the obtained results in the simulated environment as well as the real environment.

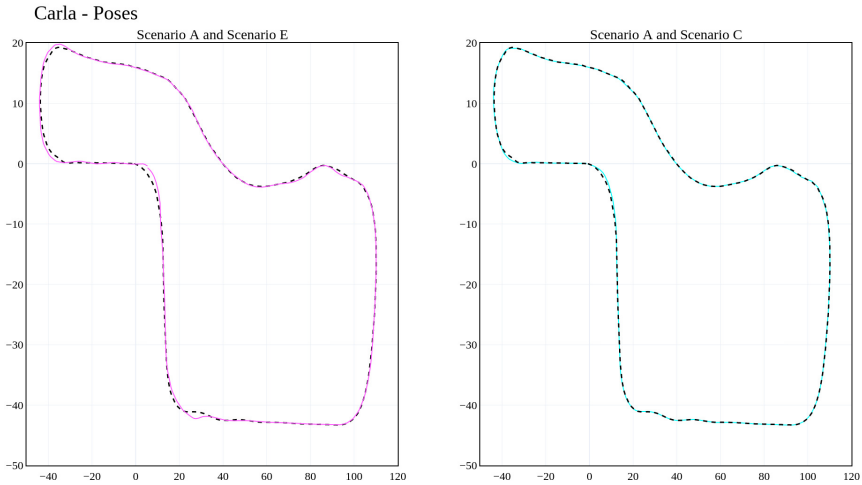


Fig. 14 The poses accuracy of the worst and best scenarios in CARLA compared with Scenario A (dotted black line).

5.3 Simulated Environment Results

In the simulation environment, Scenario B was not able to run even a single lap, not allowing us to plot it. Figure 14 shows the difference in terms of accuracy of the Scenario E and Scenario C with the Scenario A. It emphasizes the efficiency of the Qav protocol when we prioritize the localization flow instead of the interference one.

Since the time synchronization between the two Ultrascale+ is not yet accomplished, we have evaluated the tardiness between the publishing of the *Odometry* messages on the FAST-LIO node and the tardiness of their reception on the *ProxyROS2* node. The tardiness is then calculated as follows:

$$Tardiness(i) = t_i - t_{i-1} - T. \tag{3}$$

Where the sequence t_i represents the observed time while T the expected period.

Figure 15 depicts the tardiness of the evaluated scenarios. The TSN protocol introduces an overhead, yet it reduces the standard deviation since it reserves the bandwidth, making the flows more predictable. In Scenario E, we can see a strange behavior: the median value of the tardiness of the *ProxyROS2* node is less than 0. This behavior can be attributed to the default Quality of Service used for the communication between FAST-LIO and a 100-size buffer for the messages. This buffer is emptied each time an acknowledgement of the previous sent message is received. As a consequence, two consecutive messages in the *ProxyROS2* node can be received really close to each other, causing a reduction of the tardiness median value.

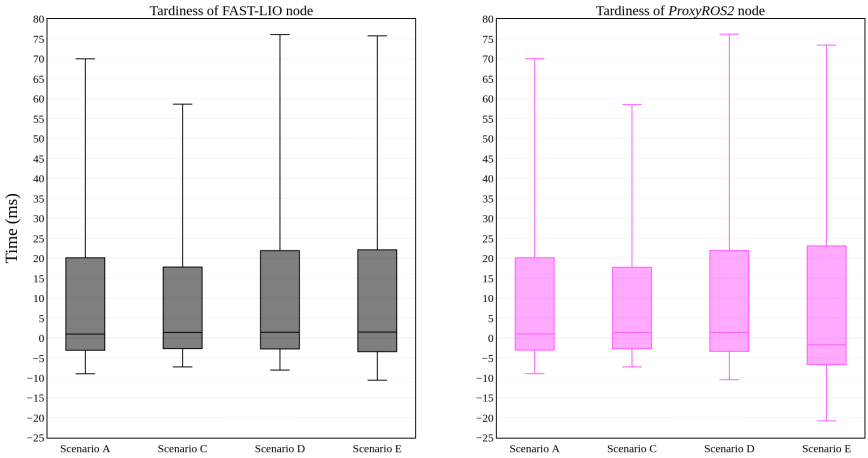


Fig. 15 Tardiness of the FAST-LIO and *ProxyROS2* nodes in CARLA.

5.4 Real Environment Results

All experiments in the real vehicle were performed on the public parking lot of the Department of Engineering at the Università degli studi di Modena e Reggio Emilia, which was controlled (no jumping pedestrians) and populated with some parked cars. As a consequence, due to the limited space, we could not test really different paths; however, we have tested and evaluated the results on the same parking lot in counterclockwise and clockwise senses, as shown respectively Figure 16 and Figure 17.

In all our tests, the designed trajectories have an oval-form and since we had to perform them for the two senses due to the presence of the already-parked cars, they can present slight differences. These figures illustrate only the last executed lap.

Due to the lower speed of the vehicle in the experiments conducted in the real environment, the car was able to execute some laps in Scenario B, allowing us to plot them. Hence, in both senses, we can clearly see the loss of accuracy in Scenario B, where the interference flow is not being controlled by the TSN protocol and the exact moment we had to stop the vehicle.

Since the trend of the tardiness is the same for both senses, Figure 18 illustrates the tardiness only for the counterclockwise trajectory.

Once again, the results demonstrate the efficiency of the TSN protocol, and moreover, we can see the behavior of the reliability of the involved Quality of Service for the communication between the FAST-LIO and the *ProxyROS2* nodes, since in scenarios A, C, D, and E, the box plots are very similar. However, the uncontrolled interference flow denies the reliability property of the QoS in Scenario B. Moreover, Scenario B presents a significant amount of packet loss, which is more than 35% between the two nodes in both senses.

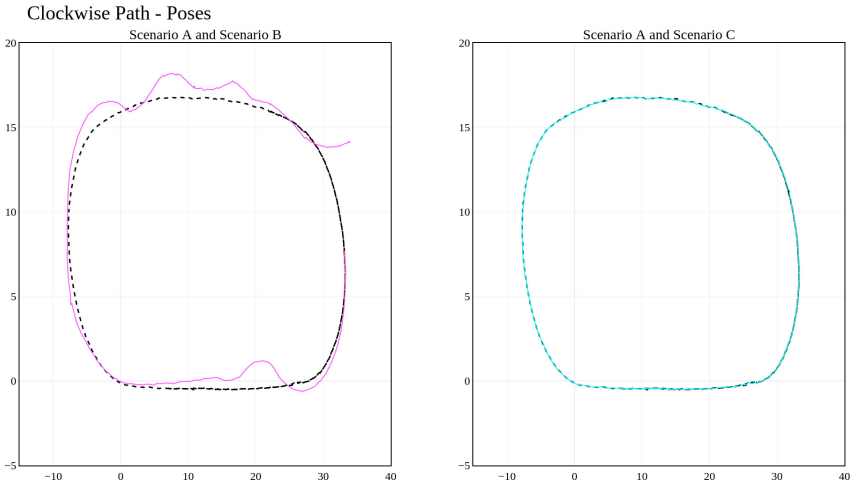


Fig. 16 The poses of the worst and best case scenarios for the clockwise path, compared with Scenario A (dotted black line).

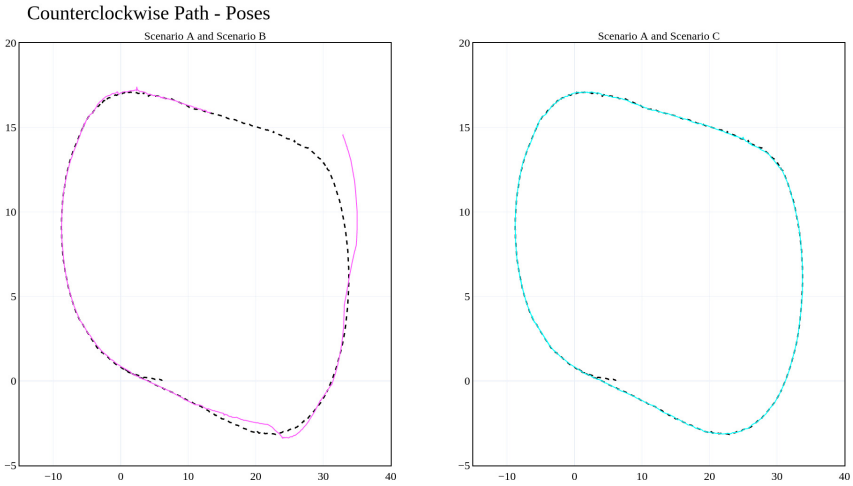


Fig. 17 The poses of the worst and best case scenarios for the counterclockwise path, compared with Scenario A (dotted black line).

By comparing scenarios A and C, the applied TSN protocol reduces the tardiness since the localization flow has a higher priority than flow 1 and flow 6. From the difference in the tardiness between the scenarios C, D, and E, it is clear that by assigning a greater *idleSlope* to the interference flow, or even both priority and *idleSlope*, the resulting tardiness increases as well. In this case, Scenario E is not experiencing the same behavior as the simulated environment due to the correct frequency of the LiDAR ($10Hz$).

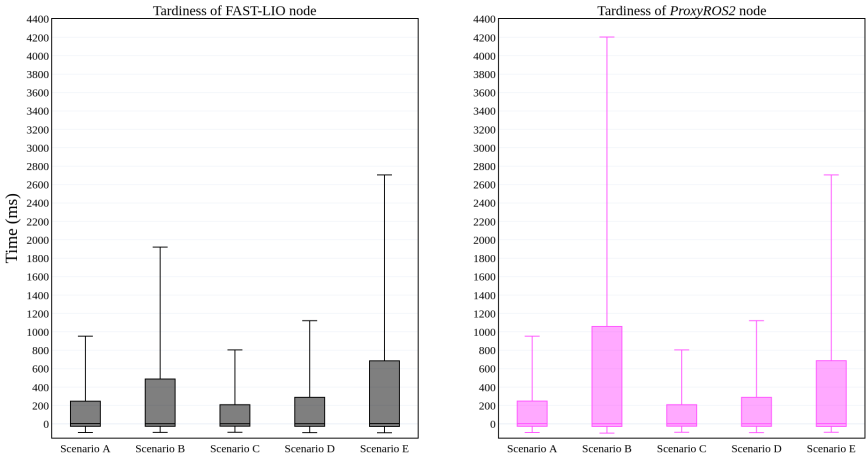


Fig. 18 Tardiness of the FAST-LIO and *ProxyROS2* nodes.

6 Current Status of the Prototype

In this section, we explain the current status of the Time-Sensitive Autonomous Architecture prototype. At the current state, we do not have a way to extract the necessary content of the *Odometry* messages in VS or in the RTOS, leading to the use of *ProxyROS2*. We are evaluating the possibility of including micro-ROS (MicroROS, 2022) in the RTOS that will use the IVSHMEM device as a custom transport layer. micro-ROS is an optimized implementation for microcontrollers that provides a ROS2 framework.

The VS solution has reached a reasonable level of stability and performance; we evaluated the tardiness in CARLA, hereafter referred to as Scenario F. This scenario was able to complete all the required 5 laps.

Figure 19 pictures the simulation environment the data flows. Once again, the involved flows are equal to the Section 5.2, the only difference is the introduction of the additional communication (VS $\longleftrightarrow$ *ProxyROS2*).

Figure 20 depicts that VS introduces an overhead; however, the resulting standard deviation is reduced. This is by virtue of the simpler and smoother environment used for the development of VS, which does not involve less predictable kernel subroutines.

7 Conclusion

The first part of this paper presents a reference architecture for next-generation autonomous systems that involves a partitioning hypervisor for enabling mixed-criticality and heterogeneous OSes (including RTOS) and real-time networking with TSN protocols.

The second part paves the way towards the deployment of VTSN-based design. We demonstrated the applicability of virtualized communication on

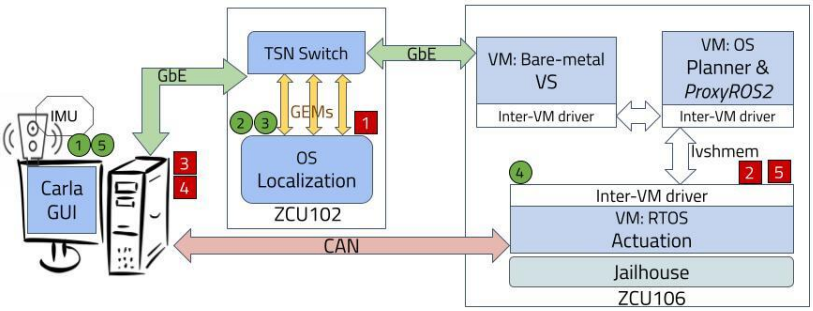


Fig. 19 Work in progress architecture and data flows for testing VS in the simulation environment. It represents the Scenario F.

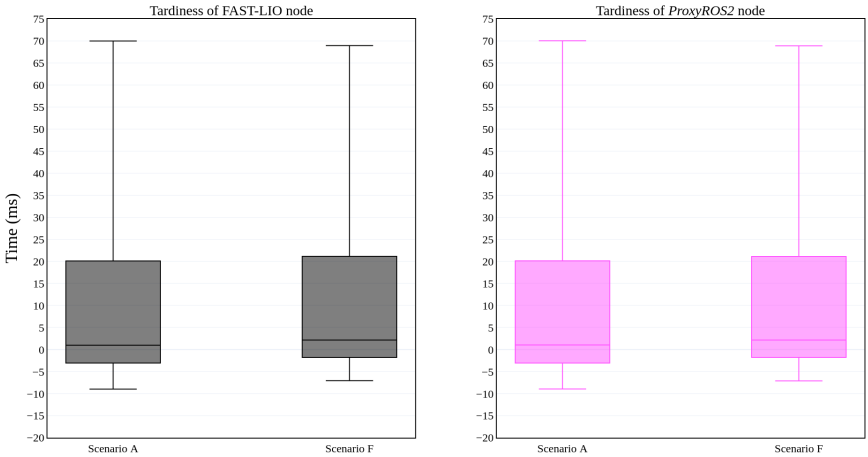


Fig. 20 Tardiness of the FAST-LIO and *ProxyROS2* nodes in CARLA.

the one hand and the applicability of TSN on the other hand by reaching performance levels that make it an adoptable solution for enabling TSN on such platforms where the number of available Ethernet ports is limited. In the future, we plan to combine these two technologies to finally provide a complete VTSN. Moreover, different aspects can be improved, e.g., reduce the number of copy operations for exchanging frames between VMs, manage the life-cycle of VMs, increase the stability of the drivers as well as the performance, use more than three VMs, add one of the TSN protocols starting from IEEE 802.1*Qav*

and port it into the R5-cluster. We believe that the VTSN mechanism will be integrated into TSAA as a possible way to manage mixed-criticality data flows, guaranteeing high determinism, as shown in Figure 2. We also plan to use the same technique to develop other kinds of autonomous systems, such as robotics in industry, e.g., (Čech et al, 2022).

The third section presents a first iteration of the proposed architecture in both a simulated and a more relevant real-world context. The architecture has been engineered, tested, and deployed with a complex autonomous vehicle application running in a dual-zone environment. We proved the feasibility of TSAA and the increase in stability in communications governed by TSN protocols.

Minerva Systems, Hipert, and Università degli studi di Modena e Reggio Emilia are still working together in order to accomplish the complete TSAA architecture, where a more complex autonomous driving system that includes the detection and uses a more robust localization algorithm, will be instantiated in the car and it will use physical and virtual TSN.

Acknowledgement

This work was partially supported by the Ministero dell'Istruzione, dell'Università e della Ricerca (MIUR) through the PRIN 2017 Program Project SPHERE — Software architecture for Predictable HEterogeneous REalTime Systems — project No 20172NNB4T and by the ECSEL Joint Undertaking (JU) under grant agreement No 101007311. The JU receives support from the European Union's Horizon 2020 research and innovation programme and Netherlands, Czech Republic, Spain, Greece, Ireland, Italy, Belgium, Latvia, Portugal, Germany, Finland, Romania, Switzerland.

References

Alderisi G, Caltabiano A, Vasta G, et al (2012) Simulative assessments of ieee 802.1 ethernet avb and time-triggered ethernet for advanced driver assistance systems and in-car infotainment. In: 2012 IEEE Vehicular Networking Conference (VNC), https://doi.org/10.1109/VNC.2012.6407430

AMD Xilinx (2020) PetaLinux Tools Documentation Reference Guide (UG1144). URL https://docs.xilinx.com/r/2020.2-English/ug1144-petalinux-tools-reference-guide

AMD Xilinx (2022a) Vivado ml overview. URL https://www.xilinx.com/products/design-tools/vivado.html

AMD Xilinx (2022b) Zynq ultrascale+ mpsoc - product tables and product selection guide. URL https://docs.xilinx.com/v/u/en-US/zynq-ultrascale-plus-product-selection-guide

Andreozzi M, Shirasat G (2022) High-performance real-time systems design from cloud to embedded edge. EasyChair Preprint no. 8064

Barham P, Dragovic B, Fraser K, et al (2003) Xen and the art of virtualization. SIGOPS Oper Syst Rev 37(5):164–177. https://doi.org/10.1145/1165389.945462

Biondi A, Casini D, Cicero G, et al (2021) Sphere: A multi-soc architecture for next-generation cyber-physical systems based on heterogeneous platforms. IEEE Access https://doi.org/10.1109/ACCESS.2021.3080842

Borgioli N, Zini M, Casini D, et al (2022) An i/o virtualization framework with i/o-related memory contention control for real-time systems. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems https://doi.org/10.1109/TCAD.2022.3202434

Boutin V, Hannart A, Essaidi A, et al (2021) Offloading autonomous vehicle machine learning algorithms to the 5g edge: A proof of concept implementation. In: 2021 IEEE 4th 5G World Forum (5GWF), pp 269–274, https://doi.org/10.1109/5GWF52925.2021.00054

Breaban G, Koedam M, Stuijk S, et al (2016) Virtualization and emulation of a can device on a multi-processor system on chip. In: 2016 5th Mediterranean Conference on Embedded Computing (MECO), pp 41–46, https://doi.org/10.1109/MECO.2016.7525767

Brilli G, Burgio P, Bertogna M (2018) Convolutional neural networks on embedded automotive platforms: A qualitative comparison. In: 2018 International Conference on High Performance Computing & Simulation, HPCS 2018, Orleans, France, July 16-20, 2018. IEEE, pp 496–499, https://doi.org/10.1109/HPCS.2018.00084

Brunner S, Roder J, Kucera M, et al (2017) Automotive e/e-architecture enhancements by usage of ethernet tsn. In: 2017 13th Workshop on Intelligent Solutions in Embedded Systems (WISES), pp 9–13, https://doi.org/10.1109/WISES.2017.7986925

Cai Y, Xu W, Zhang F (2021) ikd-tree: An incremental kd tree for robotic applications. arXiv preprint arXiv:210210808 https://doi.org/10.48550/arXiv.2102.10808

Caruso B, Leonardi L, Bello LL, et al (2021) Experimental assessment of tsn support in heterogeneous platforms with virtualization for automotive applications. In: 2021 AEIT International Conference on Electrical and Electronic Technologies for Automotive (AEIT AUTOMOTIVE), pp 1–5, https://doi.org/10.23919/AEITAUTOMOTIVE52815.2021.9662829

Cloud Native Computing Foundation (2014) Overview — kubernetes. URL https://kubernetes.io/docs/concepts/overview/

Corrigan S (2016) Introduction to the controller area network (can). Tech. rep., Texas Instruments, URL https://www.ti.com/lit/pdf/sloa101

Coulter RC (1992) Implementation of the pure pursuit path tracking algorithm. Tech. rep., Carnegie-Mellon UNIV Pittsburgh PA Robotics INST, URL https://www.ri.cmu.edu/publications/implementation-of-the-pure-pursuit-path-tracking-algorithm/

Ding P, Liu D, Shen Y, et al (2022) Edge-to-cloud intelligent vehicle-infrastructure based on 5g time-sensitive network integration. In: 2022 IEEE International Symposium on Broadband Multimedia Systems and Broadcasting (BMSB), pp 1–5, https://doi.org/10.1109/BMSB55706.2022.9828687

Dong Y, Yang X, Li J, et al (2012) High performance network virtualization with sr-iov. Journal of Parallel and Distributed Computing 72(11):1471–1480. https://doi.org/10.1016/j.jpdc.2012.01.020, communication Architectures for Scalable Systems

Dosovitskiy A, Ros G, Codevilla F, et al (2017) CARLA: An open urban driving simulator. In: Proceedings of the 1st Annual Conference on Robot Learning, pp 1–16, https://doi.org/10.48550/arXiv.1711.03938

Dugan J, Estabrook J, Ferbuson J, et al (2016) iperf - the tcp, udp and sctp network bandwidth measurement tool. URL https://iperf.fr/

eProsima (2019) Ros 2 using fast dds middleware. URL https://fast-dds.docs.eprosima.com/en/latest/fastdds/ros2/ros2.html

Evidence (2017) Erika enterprice rtos v3. URL https://www.erika-enterprise.com/

Farkas J, Bello LL, Gunther C (2018) Time-sensitive networking standards. IEEE Communications Standards Magazine 2(2):20–21. https://doi.org/10.1109/MCOMSTD.2018.8412457

Farzaneh MH, Knoll A (2017) Time-sensitive networking (tsn): An experimental setup. In: 2017 IEEE Vehicular Networking Conference (VNC), pp 23–26, https://doi.org/10.1109/VNC.2017.8275648

Fayyad H, Perneel L, Timmerman M (2013) Full and para-virtualization with xen: A performance comparison. Journal of Emerging Trends in Computing and Information Sciences Volume 10:719–727

Finn N (2022) Introduction to time-sensitive networking. IEEE Communications Standards Magazine 6(4):8–13. https://doi.org/10.1109/MCOMSTD.0004.2200046

FlexRay Consortium (2010) Flexray communications system protocol specification version 3.0.1. Tech. rep., FlexRay Consortium, URL https://svn.ipd.kit.edu/nlrp/public/FlexRay/

Fumio N, Yukinori A, Tsuneo S, et al (2022) Vehicle electronic control units for autonomous driving in safety and comfort. URL https://www.hitachi.com/rev/archive/2022/r2022_01/01c01/index.html

Garbugli A, Rosa L, Foschini L, et al (2022) A framework for tsn-enabled virtual environments for ultra-low latency 5g scenarios. In: ICC 2022 - IEEE International Conference on Communications, pp 5023–5028, https://doi.org/10.1109/ICC45855.2022.9839193

Garbugli A, Rosa L, Bujari A, et al (2023) Kubernetsn: a deterministic overlay network for time-sensitive containerized environments. arXiv preprint arXiv:230208398 https://doi.org/10.48550/arXiv.2302.08398

Gergeleit M, Streich H (1994) Implementing a distributed high-resolution real-time clock using the can-bus. In: Proceedings of the 1st International CAN Conference

Hackett E (2022) Lin protocol and physical layer requirements. Tech. rep., Texas Instruments, URL https://www.ti.com/lit/pdf/slla383

He D, Xu W, Zhang F (2023) Symbolic representation and toolkit development of iterated error-state extended kalman filters on manifolds. IEEE Transactions on Industrial Electronics pp 1–10. https://doi.org/10.1109/TIE.2023.3237872

Herber C, Richter A, Wild T, et al (2014) A network virtualization approach for performance isolation in controller area network (can). In: 2014 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS), pp 215–224, https://doi.org/10.1109/RTAS.2014.6926004

Huang YL, Lu CH (2014) A vm-based approach for real-time ethercat control. In: 2014 CACS International Automatic Control Conference (CACS 2014), pp 344–348, https://doi.org/10.1109/CACS.2014.7097214

IEEE (2018) Ieee standard for local and metropolitan area network–bridges and bridged networks. IEEE Std 8021Q-2018 (Revision of IEEE Std 8021Q-2014) pp 1–1993. https://doi.org/10.1109/IEEESTD.2018.8403927

iputils (2022) Github - iputils/iputils: The iputils package is set of small useful utilities for linux networking. URL https://github.com/iputils/iputils

Jailhouse (2015) Github - jailhouse - linux-based partitioning hypervisor. URL https://github.com/siemens/jailhouse

Jailhouse (2020) Github - siemens/linux: Linux kernel source tree - ivshmem.c file. URL https://github.com/siemens/linux/blob/jailhouse-enabling/5.4/drivers/net/ivshmem-net.c

Jansen D, Buttner H (2004) Real-time ethernet: the ethercat solution. Computing and Control Engineering 15(1):16–21. https://doi.org/10.1049/cce:20040104

Kane AA, Mariño AG, Fons F, et al (2022) Elastic gateway functional safety architecture and deployment: A case study. IEEE Access 10:91,771–91,801. https://doi.org/10.1109/ACCESS.2022.3199356

Lee J, Park S (2019) Time-sensitive network (tsn) experiment in sensor-based integrated environment for autonomous driving. Sensors 19(5). https://doi.org/10.3390/s19051111

Leonardi L, Bello LL, Patti G (2020) Towards time-sensitive networking in heterogeneous platforms with virtualization. In: 2020 25th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA), pp 1155–1158, https://doi.org/10.1109/ETFA46521.2020.9212116

Li C, Xi S, Lu C, et al (2015) Prioritizing soft real-time network traffic in virtualized hosts based on xen. In: 21st IEEE Real-Time and Embedded Technology and Applications Symposium, pp 145–156, https://doi.org/10.1109/RTAS.2015.7108436

Li C, Xi S, Lu C, et al (2022) Virtualization-aware traffic control for soft real-time network traffic on xen. IEEE/ACM Transactions on Networking https://doi.org/10.1109/TNET.2021.3114055

Liu S, Liu L, Tang J, et al (2019) Edge computing for autonomous driving: Opportunities and challenges. Proceedings of the IEEE 107(8):1697–1716. https://doi.org/10.1109/JPROC.2019.2915983

Lo Bello L, Steiner W (2019) A perspective on ieee time-sensitive networking for industrial communication and automation systems. Proceedings of the IEEE 107(6):1094–1120. https://doi.org/10.1109/JPROC.2019.2905334

Macenski S, Foote T, Gerkey B, et al (2022) Robot operating system 2: Design, architecture, and uses in the wild. Science Robotics 7(66):eabm6074. https://doi.org/10.1126/scirobotics.abm6074

Meyer P, Steinbach T, Korf F, et al (2013) Extending ieee 802.1 avb with time-triggered scheduling: A simulation study of the coexistence of synchronous and asynchronous traffic. In: 2013 IEEE Vehicular Networking Conference, pp 47–54, https://doi.org/10.1109/VNC.2013.6737589

MicroROS (2022) micro-ros — ros 2 for microcontrollers. URL https://micro.ros.org/

Mohammadpour E, Stai E, Le Boudec JY (2019) Improved credit bounds for the credit-based shaper in time-sensitive networking. IEEE Networking Letters 1(3):136–139. https://doi.org/10.1109/LNET.2019.2925176

Motika G, Weiss S (2012) Virtio network paravirtualization driver: Implementation and performance of a de-facto standard. Computer Standards & Interfaces 34(1):36–47. https://doi.org/10.1016/j.csi.2011.05.002

Obasuyi GC, Sari A (2015) Security challenges of virtualization hypervisors in virtualized hardware environment. International Journal of Communications, Network and System Sciences https://doi.org/10.4236/ijcns.2015.87026

Park C, Park S (2023) Performance evaluation of zone-based in-vehicle network architecture for autonomous vehicles. Sensors https://doi.org/10.3390/s23020669, URL https://www.mdpi.com/1424-8220/23/2/669

Qasaimeh M, Denolf K, Lo J, et al (2019) Comparing energy efficiency of cpu, gpu and fpga implementations for vision kernels. In: 2019 IEEE International Conference on Embedded Software and Systems (ICESS), pp 1–8, https://doi.org/10.1109/ICESS.2019.8782524

RedHat (1999) The newlib homepage. URL https://www.sourceware.org/newlib/

Robert Bosch GmbH (2017) E/e architecture in a connected world. URL https://www.asam.net/index.php?eID=dumpFile&t=f&f=798&token=148b5052945a466cacfe8f31c44eb22509d5aad1

ros-bridge Community (2020) Generating ring and time fields for lidars #416. URL https://github.com/carla-simulator/ros-bridge/issues/416

Shan T, Englot B, Meyers D, et al (2020) Lio-sam: Tightly-coupled lidar inertial odometry via smoothing and mapping. In: 2020 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS), pp 5135–5142, https://doi.org/10.1109/IROS45743.2020.9341176

SOAFEE (2022) Scalable Open Architecture for Embedded Edge (SOAFEE). URL https://www.soafee.io/, accessed: 2023-04-04

SoC-e (2010) Soc-e.com: System on chip & fpga ip core development. URL https://soc-e.com

Teledyne FLIR (2019) Firefly dl — teledyne flir. URL https://www.flir.co.uk/products/firefly-dl/?model=FFY-U3-16S2C-S-DL&vertical=machine+vision&segment=iis

The Institute of Electrical and Electronics Engineers (IEEE) (2010) Ieee standard for local and metropolitan area networks– virtual bridged local area networks amendment 12: Forwarding and queuing enhancements for time-sensitive streams. IEEE Std 8021Qav-2009 (Amendment to IEEE Std 8021Q-2005) pp 1–72. https://doi.org/10.1109/IEEESTD.2010.8684664

The Linux Foundation (2003) Xen project. URL https://xenproject.org/

Velodyne (2022) Velodyne - puck (vlp-16) - lidar sensor. URL https://velodynelidar.com/products/puck/

Wang J, Liu J, Kato N (2019) Networking and communications in autonomous driving: A survey. IEEE Communications Surveys & Tutorials 21(2):1243–1274. https://doi.org/10.1109/COMST.2018.2888904

Xen Project (2014) Network Throughput and Performance Guide. URL https://wiki.xenproject.org/wiki/Network_Throughput_and_Performance_Guide

Xen Project (2018) Xen Networking. URL https://wiki.xenproject.org/wiki/Xen_Networking

XSens (2022) Xsens - mti-g-710 gnss/ins - imu sensor. URL https://www.xsens.com/mti-g-710

Xu W, Zhang F (2021) Fast-lio: A fast, robust lidar-inertial odometry package by tightly-coupled iterated kalman filter. IEEE Robotics and Automation Letters 6(2):3317–3324. https://doi.org/10.1109/LRA.2021.3064227

Xu W, Cai Y, He D, et al (2022) Fast-lio2: Fast direct lidar-inertial odometry. IEEE Transactions on Robotics 38(4):2053–2073. https://doi.org/10.1109/TRO.2022.3141876

Zhao L, Pop P, Steinhorst S (2022) Quantitative performance comparison of various traffic shapers in time-sensitive networking. IEEE Transactions on Network and Service Management 19(3):2899–2928. https://doi.org/10.1109/TNSM.2022.3180160

Zheng C, Zhu Q, Xu W, et al (2022) Fast-livo: Fast and tightly-coupled sparse-direct lidar-inertial-visual odometry. In: 2022 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS), pp 4003–4009,

https://doi.org/10.1109/IROS47612.2022.9981107

Čech M, Beltman AJ, Ozols K (2022) Digital twins and ai in smart motion control applications. In: 2022 IEEE 27th International Conference on Emerging Technologies and Factory Automation (ETFA), pp 1–7, https://doi.org/10.1109/ETFA52439.2022.9921533

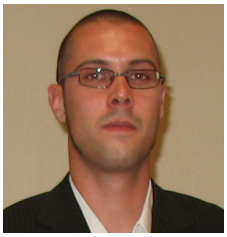
Donato Ferraro is an industrial Ph.D. student in Computer and Data Science for Technological and Social Innovation at the University of Modena and Reggio Emilia. Currently working with Minerva Systems, his research interests revolve around safety, Real-Time, and embedded systems. He is also gaining experience in capability-based operating systems.

Luca Palazzi is a master student in Computer Science at the University of Modena and Reggio Emilia, and he is working at Minerva Systems. His research is focused on hypervisors and networking for embedded systems.

Federico Gavioli is a master student in Computer Science at the University of Modena and Reggio Emilia. His main research interests are in high performance reconfigurable systems, with a particular focus on hardware-software co-design for autonomous driving applications.

Michele Guzzinati obtained his B.Sc. degree in Computer Science from the University of Modena and Reggio Emilia. He is currently employed at Hipert Srl, a spin-off of the University of Modena and Reggio Emilia. His research focuses on Robot Navigation and Autonomous Driving on Embedded and Real-Time Systems.

Andrea Bernardi is a computer scientist with expertise in high-performance computing platforms, autonomous systems, perception, and software engineering. He holds a B.Sc. in Computer Science from the University of Modena and Reggio Emilia. Currently serves as a software engineer at Hipert Srl. His research and practical experience contribute to the advancement of these fields.



Benjamin Rouxel received his M.Sc. degree (2015) and a Ph.D. degree (2018) in Computer Science, both from the University of Rennes 1 in France. After more than two years as a Post-doctorat at the University of Amsterdam, Netherlands, he is currently a Post-doctorat at the University of Modena and Reggio Emilia, Italy, in the HiPeRT Lab group. His research interests include, but are not limited to, Real-Time Systems from design concepts to concrete implementations with the addition of energy efficiency, security, and he is starting to gain interest in neural networks.

Paolo Burgio has been with HiPeRT Lab, University of Modena and Reggio Emilia, Modena, Italy, since 2014. His research interests include next-generation predictable systems based on heterogeneous many-cores and GP-GPUs, with an eye on compilers, and parallel programming models. Burgio has a Ph.D. in Electronics Engineering from the Università di Bologna, Italy, and the Université de Bretagne-Súd, France. He is a member of IEEE.

Marco Solieri is CEO and co-founder of Minerva Systems, and research fellow at the Università di Modena e Reggio Emilia. He received in 2016 a double Ph.D. in Computer Science from both the Université Sorbonne Paris Nord and the Università di Bologna, and has been also with the Université de Paris and the University of Bath. His research interests are in operating systems and hypervisors for real-time and embedded systems, with a strong accent to industrial transfer for automotive and automation verticals.