IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received 15 September 2022, accepted 6 October 2022, date of publication 10 October 2022, date of current version 20 October 2022.

Digital Object Identifier 10.1109/ACCESS.2022.3213664

RESEARCH ARTICLE

Hardware Acceleration of Data Distribution Service (DDS) for Automotive Communication and Computing

CLAUDIO SCORDINO[1], (Member, IEEE), ANGELA GONZALEZ MARIÑO[2], AND FRANCESC FONS[2], (Senior Member, IEEE)

[1]Huawei Research Center, 56124 Pisa, Italy
[2]Huawei Technologies Duesseldorf GmbH, 80992 Munich, Germany

Corresponding author: Angela Gonzalez Mariño (angela.gonzalez.marino@huawei.com)

This work was supported by Huawei Technologies Duesseldorf GmbH.

ABSTRACT The increasing growth in complexity of vehicles' functionalities is driving a technological shift in the design of software architectures in the automotive industry. Traditional signal-oriented networking is being replaced by service-oriented communications enabled by a new generation of Electronic Control Units (ECUs). The growing interest for full-fledged middlewares can be supported by these powerful ECUs. However, the new capabilities come at a non-negligible cost, which conflicts with the need to design a cost-effective solution that allows for meeting aggressive budget goals in a high volume market like automotive. In this paper, we illustrate how a significant part of the functionalities of a powerful middleware like Data Distribution Service (DDS) can be effectively implemented through hardware accelerators. We show that our approach can guarantee high performance while minimizing system complexity at the software level (e.g. AUTOSAR) by shifting painful or inefficient software implementations of QoS policies directly to hardware. This, in turn, allows to build cost-effective solutions suitable for next-generation automotive systems.

INDEX TERMS Automotive, AUTOSAR, DDS, networking, real-time, service-oriented, hardware accelerators.

I. INTRODUCTION

For decades, automotive has been a very conservative industry, with electronic functionalities made of simple Electronic Control Units (ECUs) executing tiny real-time operating systems (RTOSs) and communicating through domain-specific networks (e.g. CAN, LIN, FlexRay). The main focus has been on safety and qualification (e.g. ISO26262 [1]), while cost production has been kept under control through standardization. In particular, the AUTOSAR (AUTomotive Open System ARchitecture) partnership, started in 2004, has coordinated and driven a huge international effort to create an open and standardized software architecture for automotive ECUs. The consortium has fostered the growth of an open market where different actors (vehicle manufacturers, suppliers, service providers and companies from the electronics, semiconductor and software industry) can collaborate based on common specifications.

Fig. 1 shows an example of a traditional ECU for Gasoline and Diesel engines based on the original specifications (named AUTOSAR Classic [2]). The ECU receives a set of inputs (from the human driver, through the pedals and the cruise-control lever, from sensors or other ECUs) and controls the injection system and the throttle. The interested readers can refer to the original study [3] for further details. As illustrated in Fig. 1, at the bottom part, the AUTOSAR software stack contains the Microcontroller Abstraction Layer (MCAL). This layer, usually provided by the chip vendor, has direct access to the on-chip MCU peripheral modules and external devices and makes the upper layers independent of the specific microcontroller (MCU). An intermediate layer (called ''Basic Software'')

The associate editor coordinating the review of this manuscript and approving it for publication was Mehdi Sookhak.

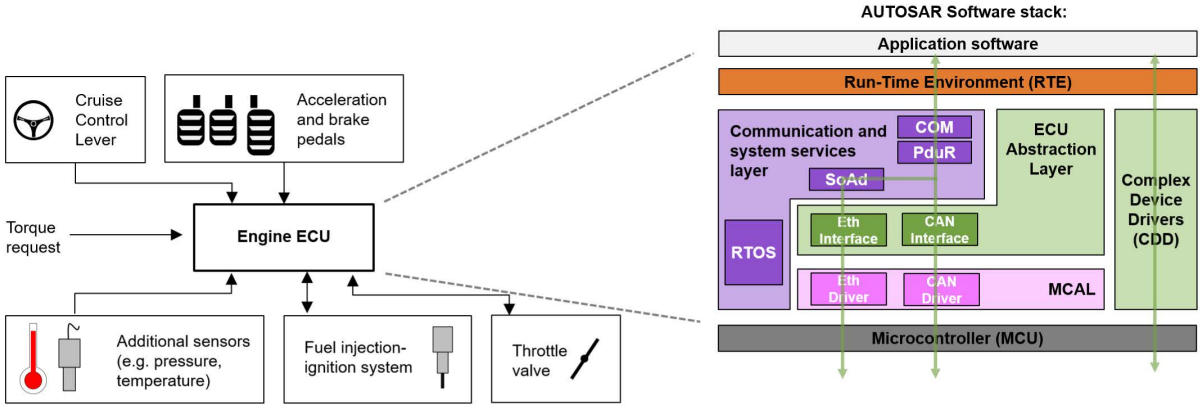

FIGURE 1. Example of automotive ECU for gasoline and diesel engines.

contains the real-time operating system (RTOS), based on the OSEK/VDX standard [4], and provides a a set of services (including communication). The generated Run-Time Environment (RTE) abstracts the communication path to the application. Finally, the functional part is implemented by the application layer consisting of a set of Software Components (SWCs).

The recent exponential increase in complexity of automotive systems [5], due to the integration of novel functionalities like assisted or autonomous driving, has shown the limits of the original specifications, calling for a revolutionary shift in the design of the computing platforms and the related software stacks [6], [7]. Current luxury cars already contain more than 100 ECUs for a total of more than 100 million lines of code [8].

To properly address this novel class of functionalities, the consortium has thus introduced an additional standard, called AUTOSAR Adaptive [9]. The software stack for this kind of ECUs consists of a general-purpose OS based on the POSIX standard (e.g. Linux) and a set of C++ libraries to support multi-thread applications. In this novel standard, the original signal-oriented paradigm has been replaced by a modern *service-oriented architecture* (SoA). Using this paradigm, the various software components are decoupled from each other and communicate by requesting and providing "services". Each component can be designed in isolation and the system is assembled by composing and integrating the various functionalities. This *separation of concerns* [10] allows to lower the complexity of the designed system to a manageable level through composability, scalability and reusability of the various components.

At the same time, automotive OEMs have started replacing traditional automotive networks with general-purpose networks (namely, Ethernet) that allow to reach higher throughputs and also solve the issue of complex cabling inside the vehicle [11], [12], [13]. Following the trend towards Ethernet networks inside the vehicle, Time Sensitive Networking (TSN) technologies are also being explored for automotive purposes [14]. TSN brings the determinism that Ethernet lacks and allows to bridge the gap towards the integration of Ethernet into real time systems such as IVNs [15].

Among the various available technologies for implementing SoA, Data Distribution Service (DDS) [16] is becoming a de-facto communication standard. The reason behind the success of DDS relies on its very powerful Quality of Service (QoS) mechanisms, that allow to set requirements at various levels of the communication stack and shape the traffic accordingly. DDS is already supported by most frameworks used for modern high-performance functionalities in automotive. These include, for example, the mentioned AUTOSAR Adaptive, the Robot Operating System (ROS2) framework [17] and its derived Apex.OS operating system [18]. In fact, according to some recent investigations [19], the ROS framework is already being used by about 80% of the automotive OEMs and Tier-1s developing autonomous vehicles. Moreover, an on-going effort aims at including the DDS support also in the more traditional AUTOSAR Classic standard [20].

DDS is not just a protocol. It is a full-fledged middleware. Therefore, it should not surprise the fact that some of its powerful functionalities come at a price, often requesting powerful hardware to be timely executed. In this paper, we illustrate how some functionalities can be effectively implemented by hardware accelerators, relieving slow microcontrollers of the execution of heavy computations and allowing the overall system to better meet the timing requirements. To the best of the authors' knowledge, this is the first attempt to propose the implementation of DDS functionalities with hardware support. Furthermore, we explore the combination of DDS with TSN technologies from a HW perspective, allowing to integrate the QoS features at different levels.

The paper is organized as follows. First, section II presents the key characteristics of the DDS middleware. Second, section III introduces the context of DDS in automotive and illustrates some automotive use-cases that can take advantage of DDS features. Section IV introduces TSN technologies in the context of automotive. Section V introduces a new HW-based network processing architecture, called

Elastic Gateway (eGW), where DDS and TSN features can be deployed, together with the Software Defined Network (SDN) implementation paradigm. Then, Section VI explains how some DDS functionalities can be effectively moved to hardware accelerators and describes the deployment of such features into the eGW architecture. Subsequently, Section VII describes the framework that allows to integrate DDS features in the HW implementation from a high level SW-based definition. Later, Section VIII explores the intersection between DDS and TSN technologies. Section IX compares our HW-centric approach with the AUTOSAR SW-centric approach and shows how our proposal is not orthogonal with AUTOSAR software stack but compatible and integrable with it. Before concluding, Section X shows a proof-of-concept of HW-accelerated DDS in eGW based on real hardware. Finally, Section XI states the conclusions and future work.

II. DATA DISTRIBUTION SERVICE (DDS)

Originally proposed in 2001, DDS became an Object Management Group (OMG) standard in 2004, with several open-source implementations available nowadays. OMG [21] is an international not-for-profit consortium producing and maintaining computer industry standards for the design of interoperable and portable systems.

The DDS specifications [16] describe a *Data-Centric Publish-Subscribe* model for distributed application communication. This model builds on the concept of a ''global data space'' contributed by publishers and accessed by subscribers: each time a publisher posts new data into this global data space, the DDS middleware propagates the information to all interested subscribers. The data-centric communication allows to decouple publishers from subscribers, thus building a very scalable and flexible architecture. The underlying *data model* specifies the set of data items, identified by ''topics''.

A **Topic** corresponds to a single data type, but it may gather multiple data-object instances (in which case, differentiated by some *key* data field). **DataWriter** is the typed object used by an application to communicate to the **Publisher** the value of data-objects of a given type. The Publisher, which can publish data of different data types, is then responsible for data distribution according to the configured QoS policies. Similarly, **Subscriber** is the object responsible for receiving data of different data types. To access the received data, the application must use a typed **DataReader** attached to the Subscriber. Thus, a *subscription* is defined by the association of a DataReader with a Subscriber. On the subscriber's side the notification can be either synchronous or asynchronous. A *domain* is a distributed concept that links all the applications able to communicate with each other. Only publishers and subscribers attached to the same domain may interact. A **DomainParticipant** represents the local membership of the application in a domain.

Fig. 2 shows a simplified vision of the interaction between a sender and a receiver belonging to the same domain. Before the communication can occur, the sender needs to instantiate

TABLE 1. Supported QoS policies of DDS version 1.4.

QoS policy	Description	DataReader	DataWriter	DomainParticipant	Publisher	Subscriber	Topic
USER_DATA	Custom user data	x	x	x			
TOPIC_DATA	Custom user data						x
GROUP_DATA	Custom user data				x	x	
DURABILITY	If data should "outlive" their writing time (e.g. late-joining DataReaders)[abc]	x	x				x
DURABILITY_SERVICE	Specifies the service implementing the durability (if any)[b]		x				x
PRESENTATION	How changes to data are presented to subscribing applications[abc]				x	x	
DEADLINE	Maximum time after which DataReader expects an update of periodic data[ac]	x	x				x
LATENCY_BUDGET	Maximum delay from data write to data reception and notification[ac]	x	x				x
OWNERSHIP	If multiple DataWriters can write the same data instance[abc]	x	x				x
OWNERSHIP_STRENGTH	Strength of the DataWriter for arbitration in case of exclusive OWNERSHIP[c]		x				
LIVELINESS	Mechanism to determine if an entity is active ("alive")[abc]	x	x				x
TIME_BASED_FILTER	Minimum time a DataReader is interested in receiving updates	x					
PARTITION	Logical partition among the topics visible by the Publisher and the Subscriber[c]				x	x	
RELIABILITY	Reliability level of message delivery[abc]	x	x				x
TRANSPORT_PRIORITY	Priority to be used on underlying transport[c]		x				x
LIFESPAN	Maximum time of validity of written data, to avoid delivery of "stale" data[c]		x				x
DESTINATION_ORDER	Logical order among changes made by Publishers to the same data instance[abc]	x	x				x
HISTORY	Behavior in case a sample changes before being communicated[b]	x	x				x
RESOURCE_LIMITS	Maximum amount of resources consumed by the service[b]	x	x				x
ENTITY_FACTORY	Behavior of an entity when creating other entities			x	x	x	
WRITER_DATA_LIFECYCLE	Behavior of DataWriter with respect to the lifecycle of the data-instances		x				
READER_DATA_LIFECYCLE	Behavior of DataReader with respect to the lifecycle of the data-instances	x					

[a] Values on the publishing and subscribing sides must be compatible.
[b] Not changeable.
[c] May appear as in-line QoS inside RTPS messages [22].

DomainParticipant, Topic, Publisher and DataWriter. Similarly, the receiver must instantiate DomainParticipant, Topic, Subscriber and DataReader. The DDS middleware will then take care of matching the two endpoints and properly deliver the sent messages. The interested readers can refer to the protocol specifications [16] for a full explanation.

It is important to note that the main DDS specification does not address the underlying transport protocol used for exchanging messages (e.g. TCP and UDP). This functionality is provided by the underlying Real Time Publish Subscribe (RTPS) wire protocol [22], specifically designed to support the unique requirements of data-distribution services. A recent Request For Proposals by OMG [23] aims at

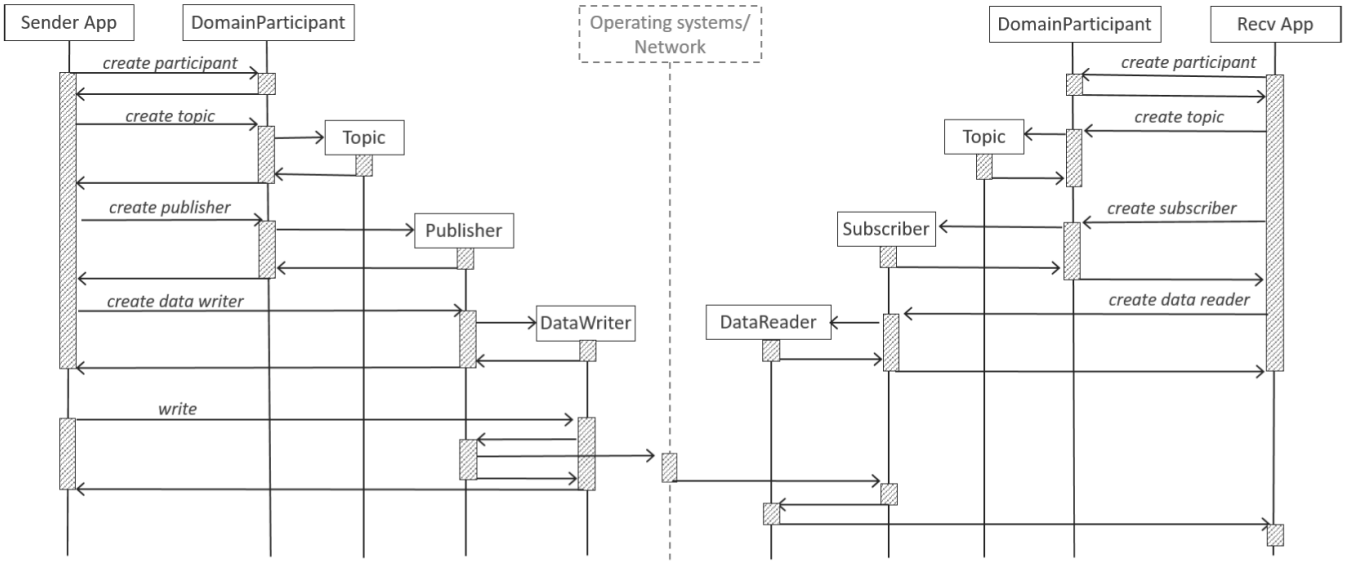

FIGURE 2. Simplified sequence diagram of DDS.

investigating the usage of DDS on top of Time-Sensitive Networks (TSN).

A. QUALITY OF SERVICE

The reason behind the success of DDS relies on its very powerful QoS mechanism, that allows to assign different *QoS policies* to the various entities in the system (i.e. Topic, DataWriter, DataReader, Publisher, Subscriber and DomainParticipant). These policies allow to control the behavior of the middleware in terms of timing predictability, overhead and resource utilization. Table 1 summarizes the possible 22 QoS policies according to the latest version of the standard (1.4). The interested readers can refer to [16] for a full description of the policies.

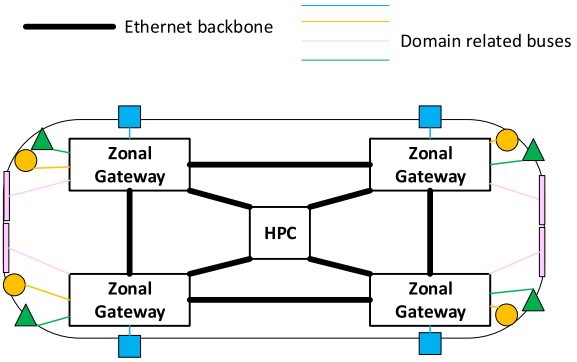

FIGURE 3. Zonal-network architecture deployment in vehicle.

III. DDS IN AUTOMOTIVE: CONTEXT AND USE-CASES

In this section, we discuss the suitability of the DDS middleware to the automotive domain. We start by introducing the state of the art of In-Vehicle Network architectures, and discuss the current challenges and opportunities regarding the design of such networks. Then, we introduce the topic of Functional Safety, which is relevant to the automotive domain, where HW-accelerated DDS capabilities can be beneficial. Later, we discuss the existing alternative to DDS in automotive: namely, SOME-IP [24]. Afterwards, we cover the typical aspects that are sometimes seen as a disadvantage of DDS for embedded systems and show how, today, it is possible to overcome them. At the end of this section, we present several use-cases for HW accelerated DDS capabilities in automotive.

A. IN-VEHICLE NETWORK ARCHITECTURE

With the introduction of novel use-cases and technologies in automotive, the electric/electronic (E/E) architecture of the vehicle is changing radically. In order to achieve the desired level of autonomy and connectivity, the number of sensors (e.g. cameras, radars, LIDAR's, ultrasound) increases dramatically. Also, the required computation becomes more complex (e.g. objects/events detection, decision making), requiring more powerful computing platforms. Apart from the cost of the new components included, the cost of the cabling required to connect them is impacting the overall cost of the solution. As described in [11], [12], and [13], the In-Vehicle Network is shifting from a logical distribution of the functionalities (i.e. domain-based architecture) to a physical distribution (i.e. zone-based architecture). Fig. 3, shows the new IVN defined for modern vehicles. The architecture is composed of several zonal gateway controllers which handle the communication with sensors and actuators of one physical area, combining the network protocols required for each of the devices. At the same time, these zonal gateways are connected between them and with the central CPU or High Performance Computer (HPC) through an Ethernet backbone.

This distributed zonal architecture simplifies the layout of sensors, actuators and cabling inside the car. However, it introduces new technical challenges. Now, each zonal gateway needs to manage different network technologies, different kinds of functionalities and traffic with different criticality levels. Authors in [25] review the challenges of current IVNs, focusing on the complexity of software configuration and mapping of functionalities to available resources. In [26], authors analyse the requirements of new ECUs in order to provide the capabilities demanded by autonomous and connected vehicles. Interested readers can find there details on performance requirements (latency, bandwidth, technologies to be supported, etc.) as well as an analysis of existing platforms in the state of the art. The conclusion of the study is that new high performance solutions are needed to reach the desired outcome, and that HW acceleration can be the key to unlock the potential of IVNs.

All in all, we see that there is a need for new strategies that allow to efficiently handle these new challenges and to enable the required performance and QoS throughout the network. Some examples are the current works on the integration of TSN in the vehicular network trying to optimize latency and reliability. Other ongoing efforts are related to safety aspects that are relevant when increasing the level of autonomy of vehicles. And on top of these functional aspects, the platforms where they run also need to keep up with the ongoing changes as seen above. In this work, we cover these trends in the following sections, and focus on a particular strategy that improves performance and QoS in the network: the HW acceleration of DDS policies.

B. FUNCTIONAL SAFETY

One characteristic of modern automotive functionalities is that they can span over multiple ECUs. When this is the case, traditional AUTOSAR Classic ECUs [2] can be used to implement real-time control loops for sensing and actuation, while AUTOSAR Adaptive ECUs can perform HPC processing. When the implemented functionality is safety-relevant, functional safety practices (e.g. ISO26262 [1]) help in lowering the risk introduced by malfunctions. For each implemented vehicle function (''item''), the safety requirements are identified (''safety goals'') and safety mechanisms are implemented to reduce the risk of hazardous events. In [27], an example of safety analysis and development of a safety concept for IVNs is presented. The benefits of HW acceleration for safety-critical features have been already explored in the state of the art. In [28], the authors propose the use of reconfigurable HW as an alternative to multi-core systems. Considering not only safety but also security features, [29] and [30] explore the benefits of HW acceleration for embedded cyber-security in automotive.

Later in this work, we show how HW-acceleration can be beneficial for the processing of DDS policies related to safety critical systems within the vehicle. The use-cases illustrated in Section III-E help in understanding how DDS can contribute to implement such safety mechanisms. For example, in case a safety goal requires to detect and handle long communication latencies, a callback can be attached to the LATENCY_BUDGET policy. Similarly, the LIVELINESS policy could be used when the safety goal requires to detect the silent failure of specific components. The DEADLINE policy allows to detect if a publisher fails to send information at the requested rate. Moreover, the OWNERSHIP policy supports the design of safe fail-over schemes. Overall, it is clear how the DDS middleware supports the design of a safe system also in the automotive domain. The alternative would be the development of the whole logic in the application code, which would become hard to port and maintain. Furthermore, being certified by a functional safety institution/authority would be also more complex, mainly in terms of proving freedom from interference (FFI) when this code gets merged with other non-safety relevant code running on the same processor or core concurrently.

C. SOME/IP VS DDS IN AUTOMOTIVE

Originally proposed by BMW, Scalable service-Oriented MiddlewarE over IP (SOME/IP) [24] is a protocol specifically designed for Ethernet-based communications in automotive. This standard specifies the serialization mechanism, the service discovery and the integration with the AUTOSAR stack. DDS, instead, is a full-fledged middleware standardized by OMG and designed as a cross-domain technology, and therefore also used for aerospace, robotics and industrial automation.

Both protocols allow distributed communication through either publish/subscribe or Remote Procedure Call (RPC) patterns. However, there are some substantial differences behind the provided functionalities:

- DDS allows looser coupling and more scalability by offering a fully-decentralized data-centric communication; the service-based pattern provided by SOME/IP, instead, is more coupled.
- On DDS, reliability and fragmentation are provided by a transport-agnostic layer (RTPS), allowing to transfer large and reliable data over (even multicast) UDP; on SOME/IP, instead, reliability needs to be provided by the underlying transport protocol (namely, TCP).
- The DDS standard also offers optional transport-agnostic security; SOME/IP, instead, does not include security functionalities and therefore relies on the functionalities offered by the underlying transports.
- DDS applications are more portable across different platforms as the specification standardizes also the full API for various programming languages.
- However, the most relevant difference is about QoS support. SOME/IP provides a very limited QoS support relying on the functionalities offered by the underlying transport. DDS, instead, offers a wide range of QoS policies (22, in the current specification as listed in

Table 1) that can be used to implement complex safety mechanisms.

For these reasons, although SOME/IP has been supported by AUTOSAR since 2014, in the recent years there has been a growing interest for DDS communications. DDS support was first included into AUTOSAR Adaptive in 2018 and now being added to AUTOSAR Classic as well [20]. Moreover, DDS is the communication mechanism behind ROS2, a framework often used for implementing novel automotive functionalities.

D. CRITICISM

In this section, we discuss some technical aspects that sometimes are used to argue that DDS is not suitable for embedded systems and/or the automotive domain. We show that, even though these issues had some foundation in the past, the technology has evolved enough to allow to overcome them.

1) CPU LOAD

One general concern when substituting lightweight automotive communication protocols (e.g. SOME/IP) with DDS is related to performance: DDS is a complex middleware and thus companies fear longer communication latencies and especially more CPU processing. Indeed, it has been shown that the execution of a DDS stack can imply a non-negligible amount of CPU processing. Bellavista et al. [31] has shown that even a scalable stack can consume up to 10% of an Intel CPU at 1.8 GHz for a simple Round-Trip-Time (RTT) test. Wu et al. [32] showed 20% of CPU usage on an Intel Xeon machine when transferring Computer Vision data for autonomous driving through DDS in the ROS2 framework. Profanter et al. [33] reported overload conditions when trying to run 100 DDS nodes on a powerful Intel i7 at 3.7 GHz [33].

The sources of this overhead come from the various QoS policies (which often imply message de-serialization and filtering) and the additional messages for service discovery [33]. Some recent work aimed at reducing overhead by implementing *zero-copy* mechanisms [19], [34]. However, the amount of CPU processing is still quite significant if compared to other technologies like SOME/IP. Indeed, the architecture proposed in this paper specifically takes into account these concerns reducing both CPU processing and latencies by moving part of the DDS stack to hardware accelerators.

2) BANDWIDTH USAGE

Some previous work [33], [35] argued that DDS has higher overhead in terms of payload size with respect to other protocols like MQTT [36], thus resulting in a higher bandwidth usage. MQTT is an OASIS standard for a lightweight publish/subscribe protocol with minimal network bandwidth requirements. However, such evaluations did not take into account the possibility of disabling dynamic service discovery of DDS, which is responsible for a significant part of the additional overhead. The evaluation showed that DDS provides ''superior performance on data latency and reliability'' than MQTT. At the same time, MQTT is not considered a viable protocol in the automotive domain due to its poor scalability [37].

3) SUITABILITY OF THE ETHERNET PROTOCOL

Another objection concerns the suitability of the Ethernet medium for in-vehicle communications. Indeed, the automotive domain traditionally preferred the design and usage of ad-hoc protocols (e.g. CAN, FlexRay, LIN) for communications. However, these technologies cannot sustain the high throughput necessary to support novel automotive functionalities. Moreover, the burden of cabling is affecting the design, the cost and the maintenance of the vehicles. For instance, the impact of its weight can compromise the autonomy of the electric vehicles (notice that the wire harness is the third heaviest and highest cost component in a vehicle, behind engine and chassis).

For these reasons, rather than designing new ad-hoc high-throughput communication mechanisms, automotive OEMs decided to switch from previous communication technologies towards Ethernet. The design of the SOME/IP protocol has been a first step towards this transition that is *already* happening in modern vehicles [38]. The next step envisioned by the AUTOSAR Consortium is the design of the zonal architecture previously illustrated, where gateway controllers communicate with a main HPC through an Ethernet backbone.

Addressing the technical and electrical suitability of Ethernet for in-vehicle communications would be an interesting discussion but out of the scope of the current work. This is being dealt with through standardization of 100/1000BASE-T1 technology migrating from Broad-Reach PHY, which was specifically designed to address the stringent electro-magnetic compatibility (EMC) requirements of Ethernet in vehicles. However, when Ethernet is used, DDS is expected to provide additional benefits over the usage of other protocols (namely, SOME/IP) as it will be illustrated in the next sections.

4) INTEROPERABILITY

Another objection is that the main benefits provided by the DDS middleware (i.e. QoS policies) are restricted only to peers communicating through this protocol. This is indeed a limitation that is being already taken into account by the AUTOSAR Consortium, which in 2022 has started an effort to make DDS available on all ECUs designed according to the standard. DDS is already officially supported on AUTOSAR Adaptive ECUs [9]. The ongoing effort, however, is standardizing the DDS support also for ECUs designed according to the more traditional AUTOSAR Classic standard [20]. Once this standardization activity will be finished, all automotive ECUs designed according to (either Classic or Adaptive) AUTOSAR or ROS2 could interoperate based on the DDS protocol, taking full advantage of its QoS policies.

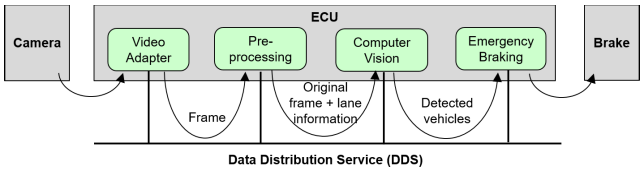

FIGURE 4. Use-case a: DDS communication in emergency braking system.

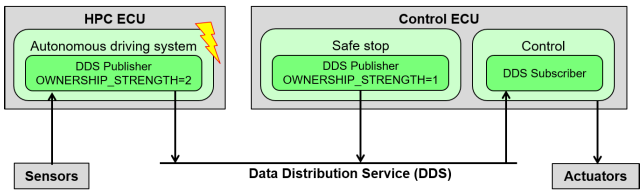

FIGURE 5. Use-case b: fail-over of autonomous driving.

E. POSSIBLE APPLICATION EXAMPLES FOR HW-ACCELERATED DDS IN AUTOMOTIVE

This section illustrates some automotive use cases that can take advantage of the features of the DDS middleware.

1) REAL-TIME SAFETY-CRITICAL COMMUNICATIONS

In automotive, as in similar real-time domains, safety-critical communications need to be delivered within a certain and bounded amount of time, otherwise some humans might be injured. As an example, Fig. 4 illustrates a camera-based Emergency Braking Assistant (EBA) similar to the one shipped with the AUTOSAR demonstrator [39]. The example consists of a pipeline of SWCs activated periodically by the operating system (with a period in the order of tens of milliseconds) and communicating through one-slot input buffers. The Pre-processing SWC identifies the current travel lane on every frame received from the Video Adapter. This information is sent along with the original frame to the Computer Vision SWC, which detects vehicles and estimates their distance. The Emergency Braking SWC, then, receives this information and decides if an emergency braking should occur. Since the system is composed of several components communicating via network, it is important to bound the E2E latency, otherwise the system might not react in time and people could be injured in a car accident.

DDS allows to assign these safety-critical communications higher priority than other communications through the TRANSPORT_PRIORITY QoS policy. This way, the middleware will take care of prioritizing this communication by e.g. increasing the priority of the underlying transport protocol.

The timing requirement, however, can be further enforced by additionally setting the LATENCY_BUDGET policy. This QoS policy, in fact, allows to specify the maximum acceptable delay from the time the data is written until the data is received and the application notified. This policy does not only help in prioritizing the messages, but it also allows to execute proper diagnostic or recovery mechanisms in case the timing requirement has not been respected.

2) HEALTHY STATE OF SAFETY-CRITICAL COMPONENTS

Collateral to communications, it is important to also check the healthy state of critical components. The LIVELINESS policy allows a safety monitor to check if a component silently fails or loses communication with the rest of the system and, in case, to trigger some recovery mechanism to enter a fail state and/or restore the failed component. In addition, the OWNERSHIP and OWNERSHIP_ STRENGTH policies allow to implement a fail-over scheme through a backup DataWriter that automatically becomes visible and substitutes the failed component.

Fig. 5 illustrates an example where, in case of failure of an autonomous driving system, another (simpler) component can take over the control and park the vehicle in safe conditions. Both the autonomous driving and the backup SWCs periodically send messages under the same DDS topic to the Control SWC which, as a result, controls the vehicle actuators. However, the SWC for autonomous driving is assigned a higher OWNERSHIP_STRENGTH than the backup SWC. This way, whenever the former SWC is alive, it ''wins'' the ownership and controls the system. However, if it silently fails, then the control gets automatically acquired by the backup SWC.

3) AVOID PROCESSING OF UNNEEDED MESSAGES

Data messages sent on the IVN could be needed by multiple subscribers, which may have different requirements about how frequently to be notified of the most recent values. Forcing all subscribers to receive (and process) the incoming messages at the highest frequency would be a waste of processing resources and might force system designers to select a more powerful and expensive hardware.

Fig. 6 shows an example where a message containing the vehicle's speed needs to be received by two different components. This is known as the ''1-N'' scenario, widely used in autonomous vehicles [32]. In the example, an AUTOSAR Classic ECU is in charge of pre-processing and publishing speed information. A high-performance ECU needs to receive the sent data at a high rate (i.e. 100 Hz) to implement autonomous driving functionalities. Another ECU, instead, needs to receive the same data at a lower rate (i.e. 10 Hz) to show the information on a dashboard.

Without DDS (e.g. using SOME/IP), the system designer would need to choose between either (i) receive and process data at a faster rate also on the dashboard or (ii) hardcode

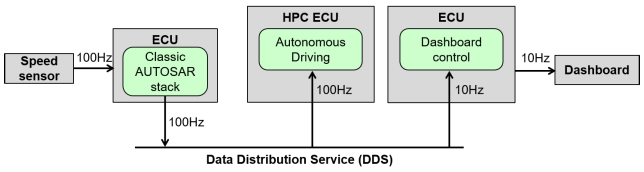

FIGURE 6. Use-case c: multi-frequency communications.

two different communications for the ECUs. DDS, instead, offers the TIME_BASED_FILTER policy that allows each DataReader to specify the minimum frequency at which data is needed. This way, each component can specify its own requirement, and the middleware takes care of dispatching data at the needed frequency. The DDS middleware running on the Dashboard will still receive data at a faster frequency, but data will be dispatched to the application at a lower frequency, thus reducing the amount of processing. Note that, implementing this functionality in hardware (as proposed in the next sections) allows to reduce CPU processing even further. In most complex scenarios, it is possible to additionally use content-based filtering, a functionality offered by DDS to specify which values of the Topic the subscriber wants to be notified for (e.g. range of values of interest).

IV. TIME SENSITIVE NETWORKING IN AUTOMOTIVE

One of the technologies to be integrated in automotive networks in order to provide determinism over the Ethernet backbone is Time Sensitive Networking (TSN). TSN is a collection of IEEE standards that define several mechanisms which allow to provide bounded jitter and latency over Ethernet networks [40]. A comprehensive survey of TSN technologies is available in [15] and [41], and the suitability of these technologies for the automotive use case is explored in [42]. TSN provides a toolkit of mechanisms to equip networks with deterministic capabilities, allowing to define the right configuration for each use case. However, this means that many parameters need to be properly defined for the network to operate correctly. The state of the art is also focusing on the configuration aspects of TSN as in [43]. In this section we provide an overview of TSN technologies applicable to the automotive domain, guided by the P802.1DG (TSN profile for automotive [44]) which is currently under development.

TSN and DDS share the same goal of providing the right QoS for frames traversing the network, but at different levels of abstraction. While TSN operates at network access layer, DDS operates at transport layer. In Section VIII we explore the intersection of TSN with DDS from a HW perspective, aiming at simplifying the software management of QoS aspects and providing the best possible performance.

A. TSN AUTOMOTIVE PROFILE: P802.1DG

Given that TSN technologies are applicable to many different industries, there are various working groups within IEEE that are working on different profiles to promote and ease the adoption of TSN in different domains. The goal of these profiles is to provide guidelines on what are the key features that can be of interest for a particular use case and simplify the selection of mechanisms in order to provide the desired performance. In automotive, the P802.1DG profile is defining the TSN features that IVNs should integrate in order to be standard compliant. The profile defines a base profile and an extended profile where some of the TSN features are required and some others are optional. An overview of the P802.1DG profile and the integrated sub-standards is given in Fig. 7. As shown in the figure, there are three main categories of the TSN sub-standards that are applicable to automotive: (i) Time Synchronization, (ii) Reliability and (iii) Bounded Latency. Next, we introduce all the sub-standards included in P802.1DG according to their functional category and provide a summary in Table 2.

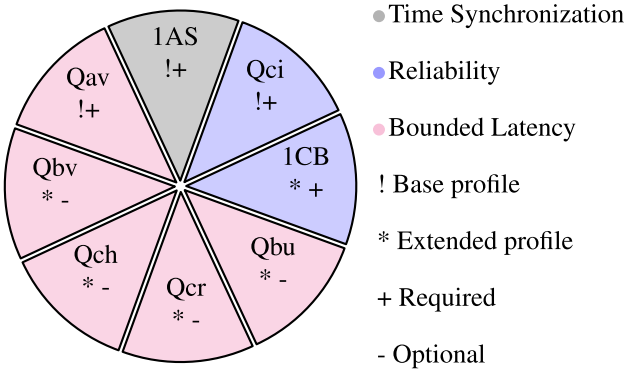

FIGURE 7. IEEE P802.1DG - TSN profile for automotive - overview.

- **Time Synchronization**: The basis of TSN technologies relies on the synchronization of the different nodes in the network, providing a common reference of time. This mechanism is standardized in IEEE802.1AS, which describes how this synchronization can be achieved [45]. For automotive, the purpose of 1AS is to provide synchronization across the whole network with a 1 μs accuracy. For this, one node acts as a master and distributes its time to the other nodes periodically, allowing the slaves to adjust their internal timing to match the one of the master.
- **Reliability**: The focus of TSN regarding reliable networks goes in two directions. On one side, it provides a mechanism to filter unwanted traffic (IEEE802.1Qci). On the other side, it provides a mechanism to ensure that critical traffic is delivered across the network by using physical redundancy (IEEE802.1CB).
  - IEEE802.1Qci [46]: This standard defines the Per-Stream Filtering and Policing strategy that can be used to filter streams that are identified to be some kind of threat. It provides mechanisms to define and identify such streams, as well as guidelines to what kind of traffic should be dropped.
  - IEEE802.1CB [47]: This standard defines the Frames Replication and Elimination for Reliability strategy that allows to duplicate traffic of critical flows in order to maximize the probability of critical messages arriving to their destination. It uses similar mechanisms as Qci in order to identify flows and process frame replication and elimination accordingly.
- **Bounded Latency**: Finally, TSN also provides several mechanisms with the aim of guaranteeing bounded latency across the network. These mechanisms can be

used independently or combined, and give flexibility to network designers to adapt them to their specific use-cases.

- IEEE802.1Qav [48]: This standard defines the Credit Based Shaper (CBS) algorithm that allows to limit the amount of bandwidth consumed by some specific flows. This is useful in limiting faulty devices or attackers that could intentionally or unintentionally flood the network resulting in severe faulty behaviour. By limiting certain high bandwidth consuming flows, it is possible to guarantee a certain bandwidth for other flows in the network, or even extra bandwidth that may be needed in case of exceptional events such as a failure.
- IEEE802.1Qbv [49]: This standard defines the Time Aware Shaper (TAS) algorithm that allows to effectively manage periodic traffic. The TAS defines a base cycle time divided in smaller time windows, and allows to control which traffic is allowed to be transmitted in each window. With this, it is possible to guarantee a certain open transmission window for critical periodic traffic, minimizing the delay experienced by this traffic.
- IEEE802.1Qch [50]: This standard defines the Cyclic Queueing and Forwarding (CQF) algorithm that allows to set an upper limit to the delay of frames. Similarly to Qbv, it defines a base cycle time, but in this case it is divided only in two windows. At the same time, all stations have two internal buffers, which are used either for transmission or reception in each window time. This way, the maximum delay of one message across the network can be limited by the cycle time and amount of hops traversed across the network.
- IEEE802.1Qcr [51]: This standard defines the Asynchronous Traffic Shaping (ATS) algorithm that provides reduced latency without the need of time synchronization. The concept is based on Urgency Based Scheduler and the implementation of a token-based algorithm to assign transmission windows.
- IEEE802.1Qbu [52]: This standard defines the strategy of frame preemption within TSN networks. The concept of frame preemption allows to define two categories of traffic: express and preemptable traffic. Given this classification, preemption defines how express traffic can interrupt the transmission of preemptable traffic, allowing to minimize the delay experienced by express traffic. Like Qcr, it is independent of time synchronization mechanisms.

V. ELASTIC GATEWAY SoC ARCHITECTURE

In this section, we introduce a novel System-on-Chip (SoC) architecture for network processing within IVNs: Elastic Gateway (eGW). This architecture has been specifically designed to meet the requirements of future IVNs defined in [26]. Elastic Gateway provides a full HW datapath composed of a set of Intellectual Property Cores (IPCores) that allow to optimize the gateway performance while keeping complexity and HW resources under control. The IPCores are optimized for the gateway application and designed with scalability and reusability in mind, allowing to create new gateway designs through the utilization of several IPCore instances. The architecture follows the Software Defined Networking approach (SDN), separating control plane from data plane although keeping them within the device and providing full configuration capabilities from the system CPU, allowing, for instance, to configure the datapath of each ingress/egress port in a different way.

TABLE 2. Summary of TSN standards included in P802.1DG.

Time Synchronization	
IEEE802.1AS	Periodic distribution of time reference through network in order to synchronize all nodes.
Reliability	
IEEE802.1Qci	Per Stream Filtering and Policing: provides mechanisms to filter undesired flows
IEEE802.1CB	Frames Replication and Elimination for Reliability: strategy to duplicate critical flows in order to make sure that they reach their destination
Bounded Latency	
IEEE802.1Qav	Traffic Shaper based on Credit Based Shaping strategy
IEEE802.1Qbv	Traffic Shaper based on Time Aware Shaper
IEEE802.1Qch	Traffic Shaper based on Cyclic Queueing and Forwarding algorithm
IEEE802.1Qcr	Traffic Shaper based on Asynchronous Traffic Shaping strategy
IEEE802.1Qbu	Support for frame preemption

The high-level architecture of eGW is depicted in Fig. 8. As illustrated in such figure, eGW is composed of three main stages, similarly to other network processing architectures: ingress stage, processing stage and egress stage. On the ingress stage, frames are received and adapted to the internal device processing format. To this aim, a new frame normalization concept and HW IPCore (Normalizer in Fig. 8) were introduced in [53]. This normalization concept allows to extract the required information from frames in order to process the frames accordingly. To do so, the normalization stage extracts the metadata of interest from the frame and composes a new frame, that we call instruction. This approach is graphically explained in Fig. 9. Traditionally, metadata is embedded in frames as a sort of new layer following the Open Systems Interconnection model (OSI) layered approach [54]in the header of each frame. However, in eGW we propose to separate the metadata from the data frame and to generate a new instruction frame that is transported in parallel with the data. This instruction frame has its own header, payload and tail, that transports the metadata related to the data frame. This new concept allows to provide full separation between control plane and data plane, since the data frame flows through the data plane, while the instruction frame carrying the metadata flows through the control plane. More details on this strategy are given in [53].

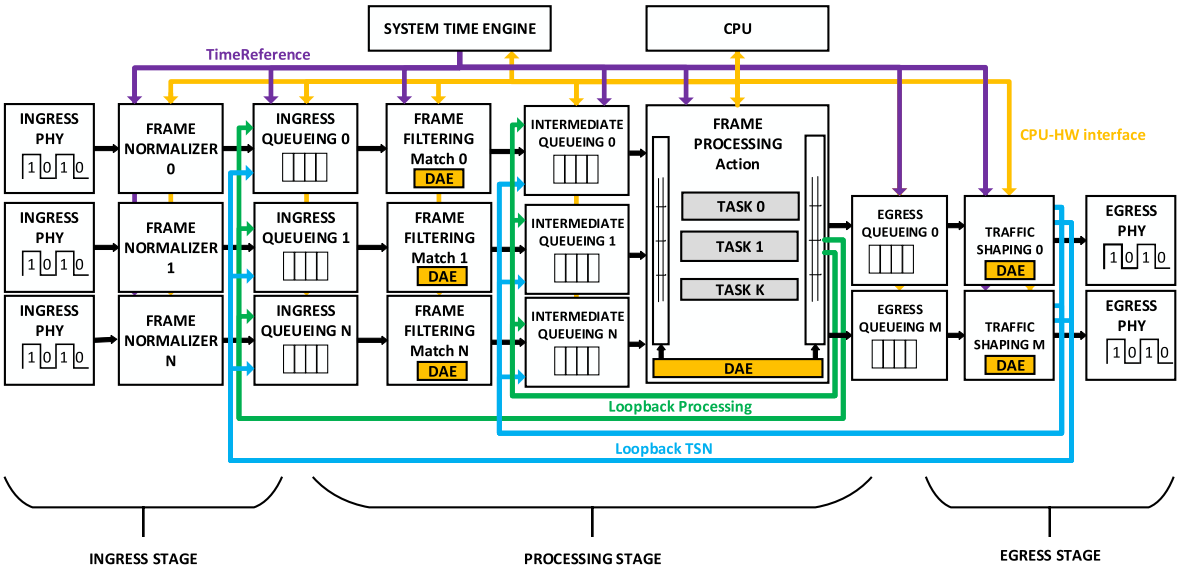

FIGURE 8. High level architecture of elastic gateway.

The format of the instruction frame is shown in Table 3. The header contains information extracted from the frame at ingress stage, such as the size, ingress port ID or type of network (CAN, Ethernet, etc.). An important field within the header is the Timestamp of the frame at ingress, which can be later used to be aware of the time that frames spend within eGW. The payload contains information about which action needs to be performed over the frame, encoded in an operation code (OPCODE), which will be used later in the processing stage. Finally, the tail consists of a checksum that allows to verify the integrity of the instruction frame across the different stages.

After normalization, frames are stored in the ingress buffers. The FIFO-based queueing buffers, which are reused across the different stages (ingress, intermediate and egress queueing in the figure) allow to accommodate frames and resolve possible conflicts on shared resources. Additionally, these buffers allow to easily change clock domains or to modify the datapath width across the different processing stages, from ingress to egress, of the SoC device. The internal architecture of these queueing buffers was described in [55].

The processing stage is composed of a single Match & Action stage with queueing modules between the steps. The match or filtering block allows to identify patterns within frames and to determine the required processing for each of them. To this aim, a Content Addressable Memory (CAM) is implemented within the filtering block. The Action stage, instead, is composed of a stack of parallel tasks which can be performed in parallel over different frames, providing maximum performance and allowing to exploit all the available HW resources. For this, the control plane of the action block determines which frame from the intermediate queueing will be assigned to each task at each moment in time. To make this decision, it takes into account the processing needed for each frame and the priority of the enqueued frames in case of conflicts. The strategy followed in order to prioritize frames is described later in this section. An example of application implemented in this architecture is detailed in [56]. In case that more than one action is required for a frame, the loopback processing path highlighted in green in the figure can be used. In this way, a pseudo-pipeline of Match & Action stages can be created, with as many stages as required by the processing of each frame. That is, the architecture allows to adapt the datapath to the processing required by each frame, optimizing thus the processing time when combined with an effective prioritization and arbitration strategy responsible for the queueing and dequeueing of the internal frames in each moment, called Distributed Arbitration Engine (DAE) and described later. More details on the use of this loopback path together with an example of application implemented on this architecture are available in [57].

After processing, frames are stored in the egress queues and finally transmitted to the egress PHY, according to the shaping rules determined by the traffic shaping module. This IPCore allows to deploy any traffic shaping standard such as the ones defined by IEEE TSN standards, or others that may

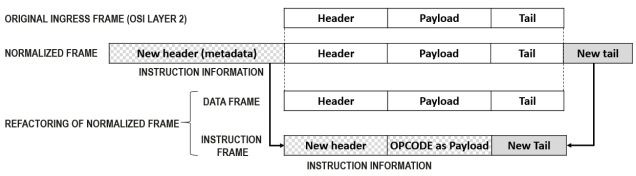

FIGURE 9. Normalization strategy.

TABLE 3. Instruction frame format.

Header					Payload	Tail
Prio	Size	Time	Port	Type	Opcode	CS

come in the future. The details of this IPCore are available in [58]. Once again, the datapath can be extended through a loopback, if required, after the traffic shaping stage. This provides the required flexibility to perform recursive processing with frames that have already reached the egress stage. More details about this loopback path and suitable use cases are described in [59].

One important aspect of the eGW architecture is the scalability and flexibility provided towards designing different families of gateway products: from a very simple gateway with few ingress/egress ports and little processing functionalities, to a high-end product with many ingress/egress ports and complex functionalities integrated, the high level architecture depicted in Fig. 8 remains the same. Exploiting this characteristics of the architecture, a system design framework called eGW Builder, which allows for selecting the geometry and capabilities of each gateway design, has been developed [60] together with a generic validation framework for the generated designs [61]. In this work, we explain how the integration of DDS features can be defined in the design framework and how this is translated into the HW implementation.

The flexibility and scalability claimed by the eGW architecture are supported by four main features: a System Time Engine (STE) that provides system time awareness, a Distributed Arbitration Engine (DAE) that enables flexible control over frames routing/processing based on dynamic priorities, an Elastic Queueing Engine (EQE) that supports flexible memory organization for the storage of frames and a Traffic Shaping Engine (TSE) that allows to shape traffic according to priorities and other shaping algorithms in place. These aspects are detailed in the following subsections.

A. SYSTEM TIME ENGINE

The System Time Engine (STE) is in charge of two structural aspects of the eGW: (i) generating the clock references to be used by the different eGW IPCores, and (ii) generating the system Time Reference. The internal architecture of STE is depicted in Fig. 10. On one hand, this block contains a Phase Locked Loop (PLL) that can be configured to generate different clock frequencies according to the configuration defined by the CPU. Then, the different clocks are routed to the corresponding IPCores, providing flexibility with regard to frequency operation of each module. For example, the ingress stage will usually operate at the rate of the ingress port, but it may be useful to operate at higher rates in the processing stage in order to accelerate complex processing tasks when required. The queueing modules in the eGW architecture allow frames to seamlessly traverse from one clock domain to another.

On the other hand, the System Time Engine includes a System Time Controller that generates the TimeReference signal within the system. This is an essential item within eGW, since it allows to distribute a common notion of time across the different eGW IPCores. As seen in Fig. 8, the TimeReference signal is distributed to all the IPCores of eGW.

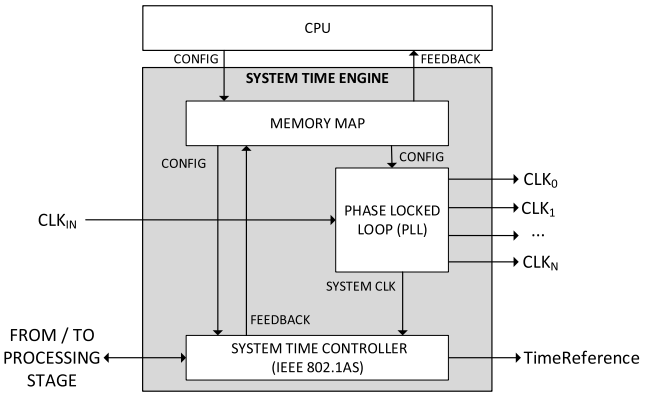

FIGURE 10. System time engine.

This information is then used at the different stages in order to perform decisions based on time. This time signal is synchronized across the network with other nodes by using the IEEE802.1AS synchronization protocol, enabling thus the integration of further TSN technologies such as synchronous traffic shaping algorithms along distributed systems composed by several nodes or ECUs like the ones depicted in Fig. 3. System Time exchanges information with the processing stage in order to extract the required time synchronization information from frames related to IEEE802.1AS.

B. DISTRIBUTED ARBITRATION ENGINE

The strategy followed by the Distributed Arbitration Engine (DAE) relies on the SDN concept at the core of eGW architecture. As explained before, eGW SoC separates control plane from data plane, allowing to handle the frame itself and the metadata related to it in parallel (instruction frame). The most important field within the context of this work is the first field in the header (see Table 3): frame priority (PRIO). This field is the one used by the DAE in order to make decisions on how to prioritize frames. This field defines the priority of the frame in real time, and is composed not only of fixed priorities that can be assigned by strategies such as VLAN tag, but also of dynamic aspects such as the time left for a frame to meet its deadline. All these bits have a meaning and weight used to determine thus how to perform the arbitration in the dequeueing. The bits within the PRIO field are organised in such a way that DAE needs only to find the bigger number among the available frames and thus select the next frame for processing/transmission (fast sorting in HW).

The DAE is used in different stages of eGW as seen in Fig. 8. Mainly, it is present in every IPCore that needs to read frames from a queueing block since this implies to decide which frame(s) to read/dequeue. The most complex arbitration happens in the processing stage where several frames can be selected in parallel and processed in different processing units as illustrated in Fig. 8 by means of the building blocks TASK 0 to K. In the egress stage, it is integrated in the Traffic Shaping Engine as described later in this section.

The composition of the PRIO field is depicted in Fig. 11. The first bit corresponds to interrupt events. This allows

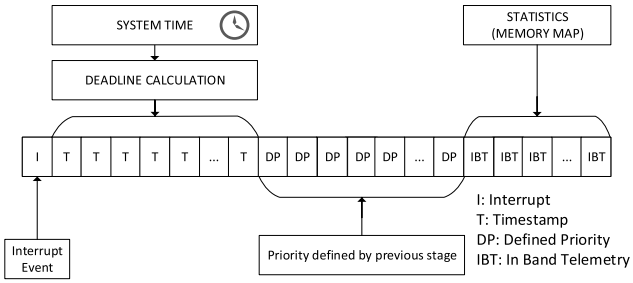

FIGURE 11. Priority field composition.

to suddenly increase the priority of a frame to the highest (because it is the most significant bit) in the case of particular events. Then, the notion of time is integrated by introducing a deadline, which indicates how much time is left for a frame to still get to its destination on time. In this case, the two's complement of the deadline is used in order to give more priority to frames with less time available. Afterwards, a predefined priority is included. This field can change across the different stages of the eGW. For instance, at the ingress stage the normalizer can assign a default priority to all the frames coming through an ingress port. Different normalizers can have different priorities, establishing a first prioritization between ingress ports. Then, in the filtering stage, the priority can change according to the pattern identified in the frame. This allows the DAE in the processing stage to know which are the frames with higher priority, including interrupts or tight timing constraints as conditions that can elevate or decrease the assigned priority at run-time. Finally, aspects coming from the internal telemetry or in band telemetry can also be part of the prioritization of frames, using information collected on the statistics module of the eGW, pursuing thus a good enough level of flexibility to define and configure such prioritization strategy.

C. ELASTIC QUEUEING ENGINE

The Elastic Queueing Engine (EQE) provides a set of FIFO memories that allow to store frames while waiting to be processed by the next stage. At the same time, it acts as interface between stages that can operate at different frequencies and/or have different datapath sizes, enabling the claimed flexibility in the eGW datapath. The internal architecture is depicted in Fig. 12. As seen in the figure, the datapath is very simple, with interconnection resources allowing to store any ingress frame into any of the internal queues. The selection of where to store the frames is done by the control plane, to be exact by the DAE, taking into account the current status of the queues as well as the metadata embedded in the instruction of each frame. Like this, the queues controller block shown in Fig. 12 is seamlessly integrated with the DAE in order to perform the queueing and dequeueing of frames at run time, in real time. On the output interface, EQE allows the next stage to read from any of the queues, providing thus maximum flexibility to manage the available frames. In terms of configuration options, EQE allows to select the number of input (i) and output (o) ports as well as the size of the internal

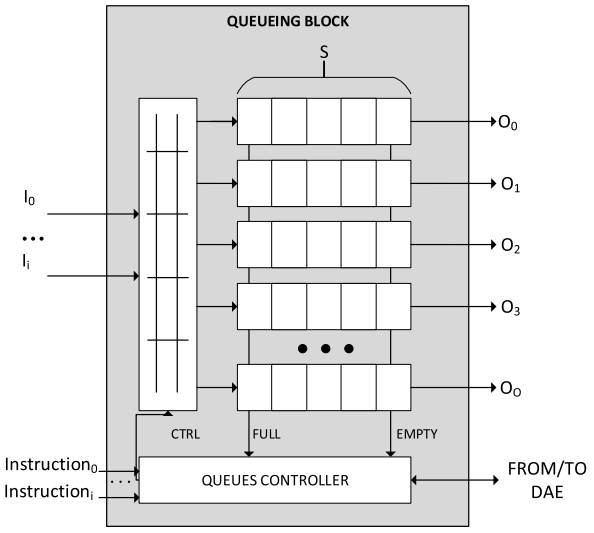

FIGURE 12. Elastic queueing engine architecture.

buffers, adapting thus to the needs of the different stages that require queueing resources within the eGW datapath.

Complementing the previously described orchestration and arbitration strategy, the queueing stage provides another extra degree of flexibility, allowing to organize the memory resources inside the queueing module in different ways. On one side, the architecture allows to define different queueing blocks based on the same structure by selecting configuration parameters such as number of FIFO modules, queueing depth per unit, input/output width, etc. On the other side, it also provides flexibility regarding the usage of the instantiated resources during operation, i.e. at run-time, allowing to maximize the use of the available memory. In other words, trying to avoid unnecessary messages drops if the eGW SoC has free space in any of the queues, independently of how they are organized, fact that does not occur in other more rigid, i.e. inelastic, chipset solutions in the market today. Further details and experimental results of some initial dynamic strategies for queueing management are available in [55]. In this work, we extend EQE by introducing a new approach to the management of queueing resources. Instead of focusing on virtually extending the queues as in the previous work, now we focus on providing a large number of small size queues that are all accessible to the next stage. With this new approach we increase the accessibility to the data stored in the queues, providing a finer grain on the decision making since many more frames can be compared and stored in parallel to improve the inline arbitration.

D. TRAFFIC SHAPING ENGINE

The Traffic Shaping Engine (TSE) allows for selecting the next frame eligible for transmission based not only on the priority field in the instruction frame, but also on other parameters defined by the traffic shaping algorithms in place. It provides a common HW architecture to implement any shaping algorithm required by TSN standards or other shaping strategies that may be of interest. It provides a set of registers

in the form of a memory map that can be configured from the CPU in order to implement different shaping strategies. Finally, it also provides the option to configure interrupt sources that can change the configuration in place, in real-time and at run-time. The high-level architecture of TSE is depicted in Fig. 13. As shown in the figure, similarly to EQE, the data plane is kept very simple allowing to minimize the latency experienced by frames traversing the eGW. The intelligence is managed in the control plane, mainly by the TSE Core. This block takes as input the configuration written in the memory map and the status of queues together with the available instruction frames. Additionally, it also has dedicated inputs for interrupt sources that may alter the behavior of the controller. In [58] we introduced the concept of TSE for the first time and performed some experiments to evaluate the capabilities to implement the different TSN standards. In this work, we extend the previous one by integrating the DAE within TSE, improving the flexibility regarding the management of frames. We also explore the suitability of TSE to support the combination of DDS and TSN with a HW-based approach (Section VIII).

The TSE core internal architecture is also visible in Fig. 13. We show how the DAE is integrated in order to decide which queue is the next one that will be transmitted. However, in this case, apart from the instruction frames available in the queueing block, DAE takes as input the output of the queue control stage, which determines whether queues are allowed to transmit or not. Previously, we saw how DAE can take the status of the queues as input and combine it with the priority field in the instruction frame. However, now we add an additional stage to the processing, determining queues eligible for transmission based not only on availability (queues status) but also on the configuration of the traffic shaping algorithms determined by the memory map. This means that for each queue we are able to determine at each moment in time whether it is allowed to transmit or not, and have DAE select the next transmission frame only among the queues for which transmission is allowed. For example, when implementing TAS or CBS, the algorithm would run individually for each queue determining whether the queue can transmit or not (i.e. gate open/closed in the case of TAS or credit available in the case of CBS). Afterwards, DAE would select the highest priority frame of the ones allowed to transmit. The main difference with the previous work in [58] is that by using DAE the priority is determined by the frames themselves, and not by the queue in which they are stored. This allows to have more options in how to store and handle the frames, giving more flexibility within the shaping stage stage and making our arbitration strategy much more scalable and HW-independent/agnostic.

VI. HARDWARE-ACCELERATED DDS: DEPLOYMENT IN ELASTIC GATEWAY

DDS has been designed to be timing predictable and efficient in its resource usage. However, the timely execution is affected by the typical issues occurring on general-purpose

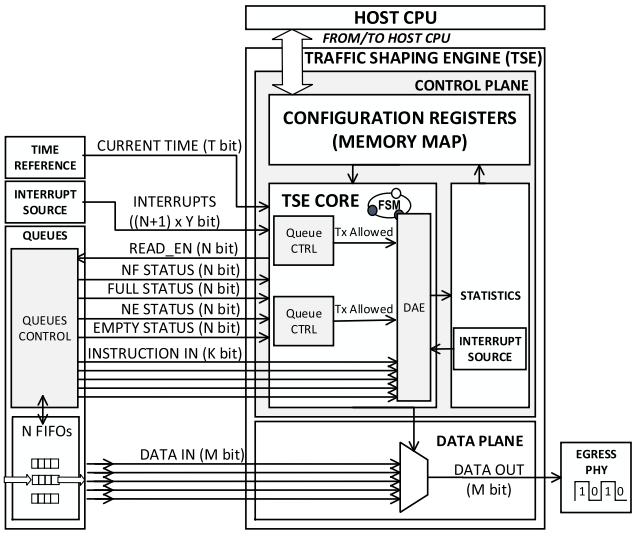

FIGURE 13. Traffic shaping engine architecture.

TABLE 4. RTPS message format.

Header	Submessage			
	Submessage Header	Submessage Element	Submessage Header	Submessage Element

platforms such as contention on shared software and hardware resources (e.g. data structures, threads, memory hierarchy, CPUs, etc.). Furthermore, being a software technology, DDS needs some non-negligible CPU processing, which might not be available on slow automotive CPUs. The community answered this issue by proposing the DDS For Extremely Resource Constrained Environments (DDS-XRCE) standard [62]. However, this profile does not offer key features of the full standard. Moreover, it introduces the need of an Agent in the software architecture to communicate with a DDS network, which could easily become a single point of failure.

It is therefore clear the need for reducing the complexity of the software stack and improving its timing accuracy to fully exploit the potential of DDS technology in the automotive domain. This section presents the novel idea of implementing a subset of the DDS QoS functionalities directly in hardware by means of reliable accelerators. Particularly, we showcase how some of the DDS QoS policies can be deployed in the previously described eGW architecture. By snooping the DDS traffic, the network equipment can automatically configure the hardware components and resources to enforce and better meet the requested QoS.

A. USE CASE 1: eGW AS A NETWORK RELAY BETWEEN PUBLISHERS AND SUBSCRIBERS

In this section, we detail how eGW supports DDS policies for traffic that is traversing the eGW — i.e., eGW is neither Publisher nor Subscriber for this traffic, just an element of the network path between them acting as a relay or hop. The main idea is to deploy the capability to detect DDS-related

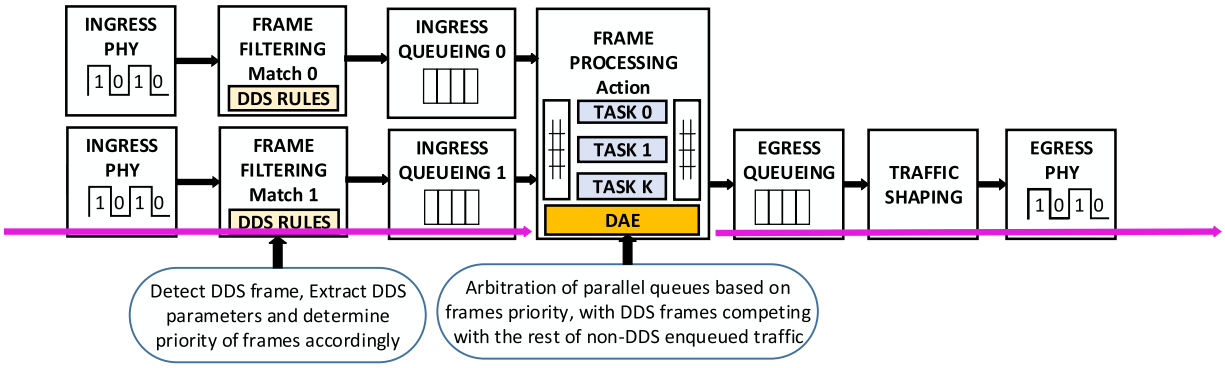

FIGURE 14. DDS support in eGW, when eGW is acting as a switch.

TABLE 5. RTPS data submessage format.

Data Submessage Header	SubmessageId
	Flags
	SubmessageLength
Data Submessage Element	EndiannessFlag
	InlineQosFlag
	DataFlag
	KeyFlag
	NonStandardPayloadFlag
	readerId
	writerId
	writerSN
	inlineQos
	serializedPayload

TABLE 6. InlineQoS parameterlist in data message format.

PARAMETER NAME	ID	TYPE
TOPIC_NAME	0x0005	String<256>
DURABILITY	0x001D	DurabilityQoSPolicy
PRESENTATION	0x0021	PresentationQoSPolicy
DEADLINE	0x0023	DeadlineQoSPolicy
LATENCY_BUDGET	0x0027	LatencyBudgetQoSPolicy
OWNERSHIP	0x001F	OwnershipQoSPolicy
OWNERSHIP_STRENGTH	0x0006	OwnershipStrengthQoSPolicy
LIVELINESS	0x001B	LivelinessQoSPolicy
PARTITION	0x0029	PartitionQoSPolicy
RELIABILITY	0x001A	ReliabilityQoSPolicy
TRANSPORT_PRIORITY	0x0049	TransportPriorityQoSPolicy
LIFESPAN	0x002B	LifespanQoSPolicy
DESTINATION_ORDER	0x0025	DestinationOrderQoSPolicy
CONTENT_FILTER_INFO	0x0055	ContentFilterInfo_t
COHERENT_SET	0x0056	SequenceNumber_t
DIRECTED_WRITE	0x0057	GUID_t
ORIGINAL_WRITER_INFO	0x0061	OriginalWriterInfo_t
GROUP_COHERENT_SET	0x0063	SequenceNumber_t
GROUP_SEQ_NUM	0x0064	SequenceNumber_t
WRITER_GROUP_INFO	0x0065	WriterGroupInfo_t
SECURE_WR._GR._INFO	0x0066	WriterGroupInfo_t
KEY_HASH	0x0070	KeyHash_t
STATUS_INFO	0x0071	StatusInfo_t

frames and configure the eGW in order to better meet the requirements of the communication during run-time. For instance, it is possible to detect DDS policies such as transport priority or latency budget, and adjust the prioritization of the traffic classes within the eGW in order to ensure that DDS requirements are met. In Fig. 14, we show how this strategy can be deployed within the eGW internal architecture. After reception, frames traverse the filtering stage where RTPS messages can be identified. Furthermore, by defining the appropriate matching rules, the information embedded in the RTPS message can also be identified, extracting the related inlineQoS parameters and deciding the prioritization of the frame within the eGW. This way, the QoS required by the services can be guaranteed both during processing stage and in the egress ports, which is where most conflicts occur.

The definition of matching rules for DDS Parameter Extraction exploits the modularity of RTPS messages, which allow to easily obtain the desired DDS parameters. The frame format of an RTPS Message consists of a fixed-size Header followed by a variable number of RTPS Submessage parts. Each Submessage also has a SubmessageHeader and a variable number of SubmessageElements, as shown in Table 4. There are several kinds of Submessages, such as Data, AckNack, Heartbeat, etc. In the case of DDS QoS policies, RTPS messages allow to send this information in a DataSubmessage. Therefore, we focus now only on the structure of this kind of Submessage. The most relevant parameters of this submessage for our work are the **inlineQosFlags** and **inlineQos**, as highlighted in Table 5. The first one signals whether any QoS property is indicated within the message. If so, the latter contains the inlineQos parameter list. The inlineQos parameter list contains information related to the DDS QoS policies used in this transaction. These can be the parameters related to any of the QoS policies marked with "c" in Table 1 together with information identifying the service and entity which the message belongs to. By exploiting this fixed structure it is possible to define the filtering rules that can extract inlineQoS parameters in a simple way. Table 6 shows the InlineQoS ParameterList format in RTPS Data messages. The parameter types are defined in the DDS specification [16]. Apart from these QoS properties, the submessage contains information related to the reader and writer involved in the communication, or other parameters outside of the scope of this work. Table 6 highlights the QoS policies that can be supported by eGW for this use-case and that will be further described in the next paragraphs.

1) DEADLINE

On automotive systems, there is often the requirement that some safety-critical data is sent and received at regular periods. An example is the radar information that needs to be sent (and received) at regular intervals to let the receiving component take decisions for emergency braking. DDS allows to codify this requirement through the DEADLINE QoS, which specifies the maximum period after which data must be sent and received, respectively on the writer's and receiver's sides.

The API to set this QoS policy for a Topic on software DDS implementations is similar to the following:

```
TopicQoS custom_qos;
DeadlineQoSPolicy deadline;
deadline.period = {\ldots, \ldots};
custom_qos.deadline(deadline);
create_topic (tp_name, tp_type, custom_qos);
```

In software implementations, however, the latency in the intermediate network equipment might trigger unwanted callback executions. In fact, even if the DataWriter respects the contract by sending the information at the right period, the DataReader might receive them with a jitter as they were not sent in time. For this reason, it is important that the intermediate network equipment is aware of the DEADLINE value to be able of prioritizing the messages when the period is expiring. This information is embedded in the RTPS messages used to send the data of a specific service from publisher to subscriber. Therefore, as seen in Fig. 14, it is possible for the eGW to extract the DEADLINE information and classify this traffic into the appropriate traffic class within the system, providing more guarantees towards meeting the required delay. Then it is just the job assigned to DAE, to perform the dequeueing of frames taking into consideration the PRIO field where all this DDS information is encoded, as shown in Fig. 11. Moreover, this policy is a good fit for interaction between DDS and TSN standards, since it could benefit from the same HW implementation as IEEE802.1Qbv, as we explain later in Section VIII.

2) LATENCY BUDGET

The LATENCY_BUDGET QoS policy specifies the maximum delay from data write to data reception and notification.

The API to set this QoS policy for a Topic on software DDS implementations is similar to the following:

```
TopicQoS custom_qos;
LatencyBudgetQoSPolicy latency;
latency.duration = {\ldots, \ldots };
custom_qos.latency_budget(latency);
create_topic (tp_name, tp_type, custom_qos);
```

Similarly to the DEADLINE policy, this information comes in a RTPS message when exchanging information between publisher and subscriber. Again, eGW can extract this information and prioritize the traffic accordingly. In this case, classification could be based on a set of thresholds or ranges defined within the DDS rules (when latency budget < X, then priority–>X, when latency budget > Y, then priority–> Y). Since the latency budget format is defined by the RTPS frame, it is possible to define the HW accordingly in order to be able to guarantee that all values of the policy can be handled in the HW. Furthermore, thanks to the SDN approach followed within eGW, these thresholds can be updated by the CPU when required, allowing to dynamically change the configuration used to prioritize DDS traffic.

3) TRANSPORT PRIORITY

The TRANSPORT_PRIORITY QoS policy specifies the priority to be used on underlying transport.

The API to set this QoS policy for a Topic on software DDS implementations is similar to the following:

```
TopicQoS custom_qos;
TransportPriorityQoSPolicy priority;
priority.value = \ldots
custom_qos.transport_priority(priority);
create_topic (tp_name, tp_type, custom_qos);
```

As in the previous cases, the DDS Parameter Extraction module gets this information and defines the priority used for these frames in the egress stage. In this case, the DDS policy is already enforcing an explicit priority, which only needs to be converted to the internal priorities in order to assign it to the corresponding frame. The mapping between priorities is done based on a table defined by the eGW CPU, which guarantees that every possible priority in DDS will have a corresponding priority in eGW.

4) RELIABILITY

The RELIABILITY QoS policy specifies the reliability level of message delivery.

The API to set this QoS policy for a Topic on software DDS implementations is similar to the following:

```
TopicQoS custom_qos;
ReliabilityQoSPolicy rel;
rel.kind = RELIABLE_RELIABILITY_QOS;
custom_qos.reliability(rel);
create_topic (tp_name, tp_type, custom_qos);
```

In this case, the information can be identified and used to prioritize traffic requiring higher reliability, but also to configure other internal reliability mechanisms, such as Frames Replication and Elimination for Reliability (FRER) strategy, defined in TSN standards. The interaction between this DDS policy and TSN standard are described in Section VIII.

B. USE CASE 2: eGW AS A PROCESSING ELEMENT HOSTING A PUBLISHER OR SUBSCRIBER

In this section, we describe how eGW can support DDS QoS policies for traffic that is generated/consumed at the eGW itself — i.e. the case when eGW hosts the application of the reader/writer. Essentially, new HW accelerators are deployed in the frame processing stage where some of the DDS QoS policies can be offloaded from the CPU, as shown in Fig. 15. On one side, there is HW support for publisher features, which collect information from the CPU memory

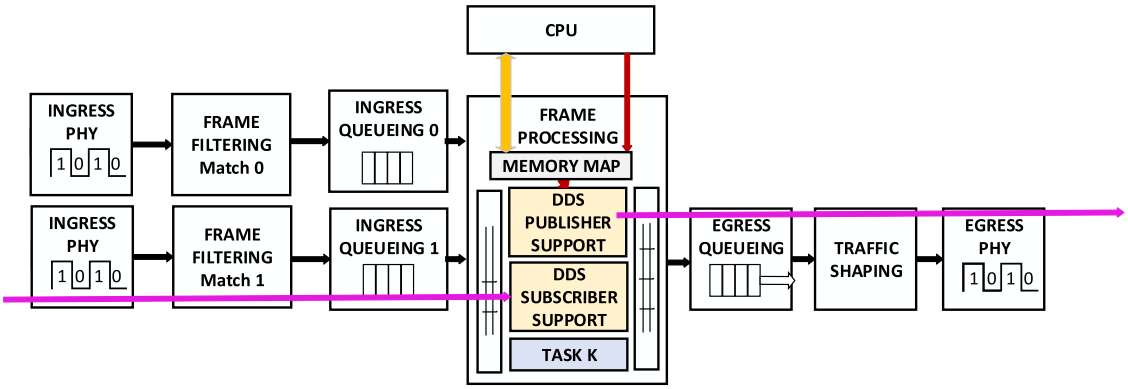

FIGURE 15. DDS support in eGW when as publisher/subscriber.

map and generate the required traffic (e.g. alive messages). On the other side, there is HW support for subscriber features, where traffic is first analyzed in the HW, some characteristics are extracted, and then frames are sent to the CPU when applicable.

Next, we elaborate on some DDS policies that can be deployed within the described architecture and how this can be implemented.

1) LIVELINESS

The LIVELINESS QoS policy describes a "mechanism to determine if an entity is active ("alive")".

The API to set this QoS policy for a Topic on software DDS implementations is similar to the following:

```
TopicQoS custom_qos;
LivelinessQoSPolicy live;
live.kind = AUTOMATIC_LIVELINESS_QOS;
live.duration = {\ldots, \ldots };
custom_qos.liveliness(live);
create_topic (tp_name, tp_type, custom_qos);
```

When configured, entities send alive messages informing that their instance is up and running. The policy can be configured to either let the middleware send these messages automatically (as in the example above) or to leave this responsibility to the application code.

For this policy, a different type of RTPS message is used: Heartbeat. As highlighted in Table 7, within the Heartbeat message, the LivelinessFlag is the element that needs to be asserted in order to inform the reader about the writer status. The other fields can also be filled accordingly to maximize usability of this Heartbeat message, although the purpose of each field is not of interest for this particular work. In our case, eGW can easily perform the task of periodically sending Heartbeat messages within the Publisher Support module, offloading the CPU of this processing.

2) TIME-BASED FILTER

The TIME_BASED_FILTER QoS policy defines the minimum time a DataReader is interested in receiving updates.

TABLE 7. RTPS heartbeat submessage format.

Heartbeat Submessage Header	SubmessageId
	Flags
	SubmessageLength
Heartbeat Submessage Element	EndiannessFlag
	FinalFlag
	LivelinessFlag
	GroupInfoFlag
	readerId
	writerId
	firstSN
	lastSN
	count
	currentGSN
	firstGSN
	lastGSN
	writerSet
	secureWriterSet

The API to set this QoS policy for a DataReader on software DDS implementations is similar to the following:

```
DataReaderQos custom_qos;
TimeBasedFilterQosPolicy time;
time.minimum_separation = {\ldots, \ldots };
custom_qos.time_based_filter(time);
create_datareader(tp_name, custom_qos, \ldots );
```

When established, Readers filter frames which are outside of the receiver window (i.e. earlier than what the time-based filter requires). However, this policy is not embedded as an inlineQoS parameter within an RTPS message. In this case, it is a configuration that belongs to the reader and can be specified in the device supporting the reader application processing. eGW supports this feature within the Subscriber support module just dropping messages according to the time window rules, offloading thus the CPU from this filtering task and reducing unnecessary workload.

3) DESTINATION ORDER

The DESTINATION_ORDER QoS policy defines the logical order among changes made by Publishers to the same data instance. In essence, it defines means to resolve conflicts when several publishers update the same data instance. For

example, it allows to choose recording values based on reception timestamp (last received prevails) or source timestamp (last sent prevails). The API to set this QoS policy for a Topic on software DDS implementations is similar to the following:

```
TopicQoS custom_qos;
DestinationOrderQoSPolicy order;
order.kind = \\
BY_RECEPTION_TIMESTAMP_DESTINATIONORDER_QOS;
custom_qos.destination_order(order);
create_topic (tp_name, tp_type, custom_qos);
```

Typically, the DDS_BY_SOURCE_TIMESTAMP_ DESTINATION_ORDER_QOS policy is used, for which a timestamp must be included in the messages by every Writer. In eGW, this can be supported in the subscriber support module, delivering the chosen data to the CPU and abstracting from the complexity of such conflicts.

All in all, the examples elaborated in this section prove how we can reduce the software complexity of certain timing-related functions when appropriately ported to a dedicated and cost-effective hardware architecture based on hardware accelerators (HWA). Our proposal supports all this processing performed in-memory and inline in a smooth and more reliable manner, skipping by design those potential software uncertainties/interferences that appear when deploying in a multicore or multi-thread SW-centric solution. In summary, our technical proposal consists in moving all this time-critical processing from upper SW layers to a lower layer managed in HW but configurable in SW by the host CPU. With this approach, all that timing complexity and uncertainty is gone, just by design.

VII. FROM DDS POLICIES TO IPCores: ELASTIC GATEWAY BUILDER TOOL

The composition of a complex HW design is often an arduous task. To reduce the complexity, there are several frameworks available targeting different applications. For instance, authors in [63] present a framework that eases the integration of Deep Neural Networks in FPGAs. In [64] a framework to automate the deployment of TSN switches configuration in FPGAs is presented. Following this trend, we briefly introduce the automation framework that enables the design of gateways based on eGW architecture. More details on this framework are available in [60] and [61].

As seen before, eGW allows to instantiate different instances of a GW design by choosing which features are integrated in each of them. To ease this task, an automation design framework, ''Elastic Gateway Builder'' has been developed [60]. The framework allows to first define the geometry and shape of the gateway design (architecture) and later configure the embedded functions and features of each IPCore (micro-architecture), both levels managed by means of parameters that are configurable. Fig. 16 shows how the framework allows to select the DDS policies that will be included in the design, and how these are mapped to the HW implementation.

TABLE 8. eGW DDS policies memory map.

DDS Policy	Parameter	Type
LIVELINESS	Alive timeout	Natural
TIME_BASED_FILTER	Minimum Time	Natural
DESTINATION_ORDER	Rx vs. Tx Timestamp	Boolean

In the top of the figure, an architecture example with two ingress and two egress ports is shown. After defining this first step, the user can go block by block configuring the micro-architecture parameters. For instance, when configuring the micro-architecture view of the Action block, there are several sub-blocks available, as seen in the left bottom of the figure, corresponding to the low-level components of the Action stage. When the selection of parameters of all blocks is done, the designer can click on the ''Generate HDL'' button, to automatically generate the code corresponding to this particular design. In this automatically generated code, the IPCores of eGW will include the required blocks according to the parameters selected, as seen in right bottom corner of the figure. In this case, the publisher and subscriber support for DDS features will be included. For more details on how the code is automatically generated, interested readers can refer to [60].

Overall, Fig. 16 shows the graphical interface of the tool that allows to select the desired features, and how the selection is later translated into the HW implementation, by instantiating the corresponding IPCores from our eGW Builder library. In this example, we focus on the DDS features, particularly the Liveliness policy. Additionally, for each of the selected features, the associated registers are automatically introduced in the memory map of the full chipset, allowing to configure the required functionalities. An example of the registers corresponding to the DDS features instantiated in Fig. 16 is shown in Table 8.

With this, we show how the gap between high-level DDS policies and low-level HW IPCores can be bridged thanks to the SDN strategy followed in eGW architecture and supported by eGW Builder as automation tool.

VIII. COEXISTENCE OF DDS WITH TSN IN eGW

The traditional layered approach defined by the OSI model is the de-facto standard in networking devices. Its success resides on the fact that it allows to decouple functionality from transport and physical layers, permitting to reuse functionalities across different network technologies and increasing the scalability of network management. It also simplifies the application layer processing, by allowing higher layers to operate without the need of knowing anything about how the lower layers work. However, this freedom in terms of functionality comes at a cost: it is not only that higher layers do not need to know how lower layers operate; it also means that they cannot influence their behavior, even if they want to.

For this reason, QoS management is nowadays commonly deployed in lower layers (TSN at network access layer) but there is no link to the application layer. On the other side,

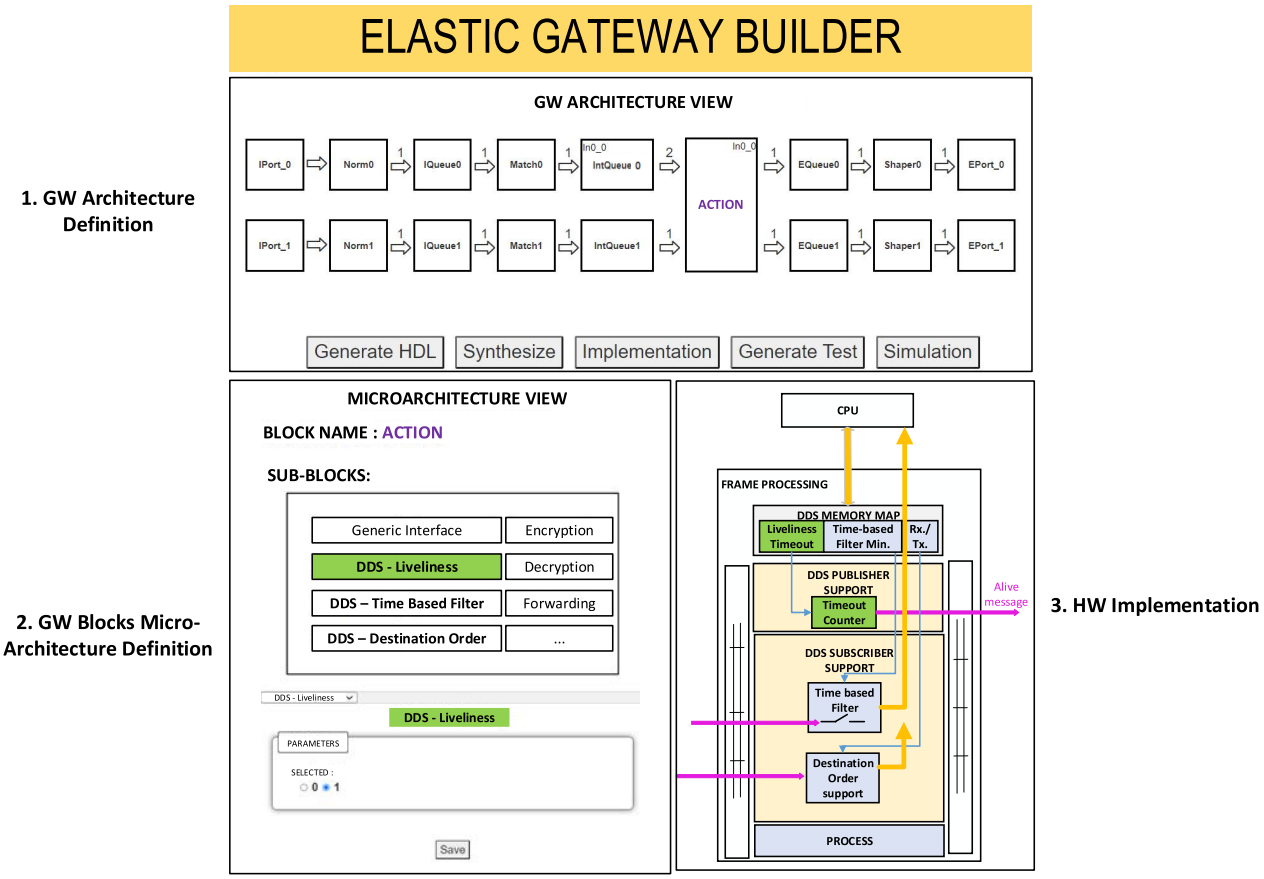

FIGURE 16. From DDS policies to HW implementation with eGW builder tool.

TABLE 9. Mapping of QoS features to simplified OSI model.

Layer	QoS features
Application Layer	-
Transport Layer	DDS library
Protocol Layer	RTPS protocol
Network Access Layer	TSN technologies

middlewares such as DDS try to bridge this limitation by offering service related QoS features at transport layer. However, without linking DDS to the lower layers, the deployment is still subject to how the link and physical layers are managed. Table 9 shows how the existing QoS mechanisms are mapped to the different layers defined in OSI model (a simplified view).

There are already some works in the literature exploring the convenient integration of TSN and DDS. In [65] authors exploit the QoS capabilities of DDS in terms of traffic prioritization to guarantee the QoS of their application, while using TSN to provide time synchronization across the network. However, in this research TSN and DDS just happen to be in place in the experiments, but are independent of one another. In [66] authors describe a case study of DDS over TSN for military applications where they map the DDS applications to TSN enabled end stations. Similarly to the previous reference TSN and DDS run in parallel, but independently. In [67], authors explore the capabilities in terms of QoS of wireless TSN networks when combined with DDS. In such work, the intersection between TSN and DDS is introduced by mapping DDS topics to TSN streams.

In here, we identify a gap in the state of the art, which is why we explore how DDS and TSN QoS features can be seamlessly merged or combined in order to provide this missing bridge that would allow to define low-level QoS properties at higher levels of abstraction. This bridge is an important advance in the state of the art since it allows to ensure the performance of application level services that have highly stringent real-time requirements for their operation.

In this section, we analyze how DDS and TSN can be successfully combined in order to improve the QoS management from three different perspectives. Moreover, we also show how this combination of DDS and TSN can be deployed in the previously introduced eGW architecture, maximizing performance through the combination of DDS, TSN and HW-based network processing. Table 10 summarizes the DDS policies that can be mapped to TSN technologies and how they are deployed in eGW.

A. TRAFFIC PRIORITIZATION: WHO GOES FIRST?

The whole problem of traffic management can be abstracted as a conflict resolution problem. In the end, QoS strategies

TABLE 10. Mapping of DDS policies to TSN technologies and eGW SoC.

DDS policy	TSN feature	eGW parameters
TRANSPORT_PRIORITY	Strict Priority	Frame Priority Field
LATENCY_BUDGET	Strict Priority	Frame Priority Field
DEADLINE	IEEE 802.1Qbv	TSE Configuration
RELIABILITY	IEEE 802.1CB	FRER Support

are needed because frames encounter conflicts along their trip through the network. In other words, in a network without conflicts, no QoS strategies would be required. However, conflicts do exist in real-world networks, mainly due to the presence of shared resources that need to be arbitrated, which is why different strategies are needed in order to solve them. The first approach to tackle this problem is to introduce traffic prioritization, i.e. out of a collection of frames that may be eligible for transmission at a specific moment in time, ''Which one should go first?'' Defining priorities allows to make this decision in a simple way.

DDS provides means to prioritize traffic through 2 different QoS policies. The first one is the TRANSPORT_PRIORITY policy, which explicitly indicates which should be the priority used for a particular frame at transport layer. Another policy that expresses priority in an indirect way is the LATENCY_BUDGET policy. By informing about the available time that a frame has to reach its destination, dynamism is achieved with regard to the priority of the frame: if a frame has a wide margin to arrive to its destination, a low priority can be used for its transport, giving more priority to other frames; if, instead, it is running out of time, the priority of this particular frame can be increased, allowing it to meet the latency requirement.

TSN also allows to classify traffic according to priorities. In this case, frames are divided into different Traffic Classes (TC) (8 according to IEEE802.1Q) with each TC having a priority assigned. Then, a Strict Priority scheduler chooses the next frame for transmission, which will be the one with highest priority out of the eligible frames in a particular moment in time. In here we see an opportunity to make a bridge between the DDS specification and TSN, matching DDS prioritization to TSN traffic classes. For the TRANSPORT_PRIORITY policy, this is a direct match between the priorities specified by DDS frames and the defined TC within the gateway, which can be configured in the gateway memory map. For LATENCY_BUDGET, instead, a dynamic range can be defined, such that depending on the remaining time that a frame has to achieve its destination, the priority used at L2 level (i.e. the traffic class) can be adjusted. Table 10 shows how the mapping between DDS policies, TSN traffic classes and eGW registers in the memory map can be deployed.

B. PERIODIC TRAFFIC MANAGEMENT: A COMMON CONCEPT OF PERIOD, TIME AND DEADLINES

Another common challenge of traffic management is how to efficiently deal with periodic traffic. On one side, periodic traffic should be, at first glance, easy to manage since we know in advance when it is going to occur. However, when mixed with traffic sent with different periodicity and also with event-based traffic, the correct management is not so simple. Furthermore, in order to be able to make a good planning of resources for periodic traffic, a common notion of time needs to be shared across the different devices of the network. This means, that in order to plan for a flow that may come at a specific moment in time, all nodes must have a common knowledge of ''what time it is''. In other words, all nodes need to be synchronized.

For this, TSN offers the IEEE802.1AS standard that allows to distribute the time of a master node so that all the other nodes can synchronize with it with an accuracy of 1 μs or less. Once synchronization is in place, different strategies can be used in order to handle periodic traffic. The proposal from TSN technologies is IEEE802.1Qbv, also known as Time Aware Shaper (TAS). Basically, TAS defines time windows in which only one traffic class is allowed to transmit, ensuring that periodic traffic will encounter an open path when arriving to the node, minimizing thus the end-to-end latency if all nodes synchronize these windows properly.

On the DDS side, the DEADLINE policy defines the maximum time (i.e. the period) between updates of a data instance. Again, we can match the period of the services with a particular deadline policy and make it a part of the TAS cyclic traffic management, ensuring that the traffic corresponding to a particular service will always meet its deadline. This can be deployed in eGW by appropriately configuring the registers within the traffic shaping stage that allow to define the time windows for each traffic class, as seen in Section V.

C. TRAFFIC RELIABILITY MANAGEMENT: WHEN BEST EFFORT IS NOT ENOUGH

Apart from timing constraints, QoS also applies to the reliability of frames transmission. In safety related applications this is of utmost importance since people's health condition might be at stake. Most of the time, reliability requires redundancy. This is because safety related information usually needs to be received within a bounded amount of time which does not allow for data re-transmission, and also because the system must be robust to a link failure where no re-transmissions would be feasible as in best-effort traffic. There are several strategies for redundancy, ranging from topology decisions where the amount of redundant links are chosen, to protocol level strategies deciding which traffic should be replicated, how and when.

DDS provides the RELIABILITY policy that allows to specify the reliability required for a particular service. However, this is more a high-level requirement than an implementation specification, since DDS does not infer how this reliability should be provided.

TSN also considers the topic of reliability for which IEEE802.1CB has been defined. This standard defines the ''Frames Replication and Elimination for Reliability'' algorithm (FRER), which provides a standard way to provide

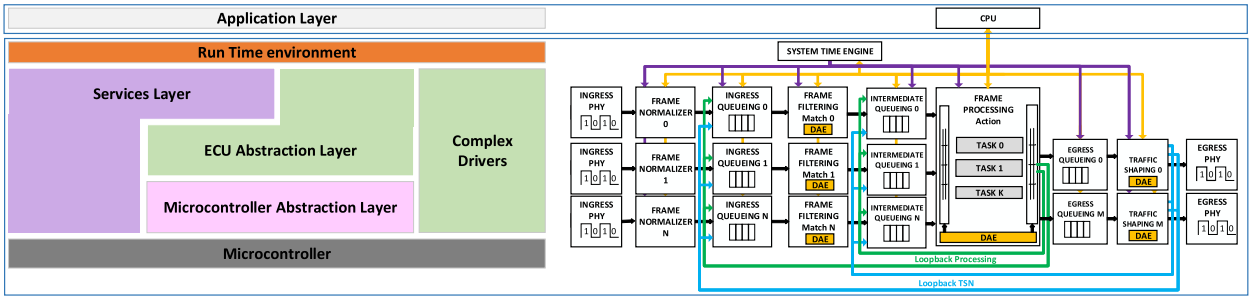

FIGURE 17. AUTOSAR SW stack vs eGW SoC architecture.

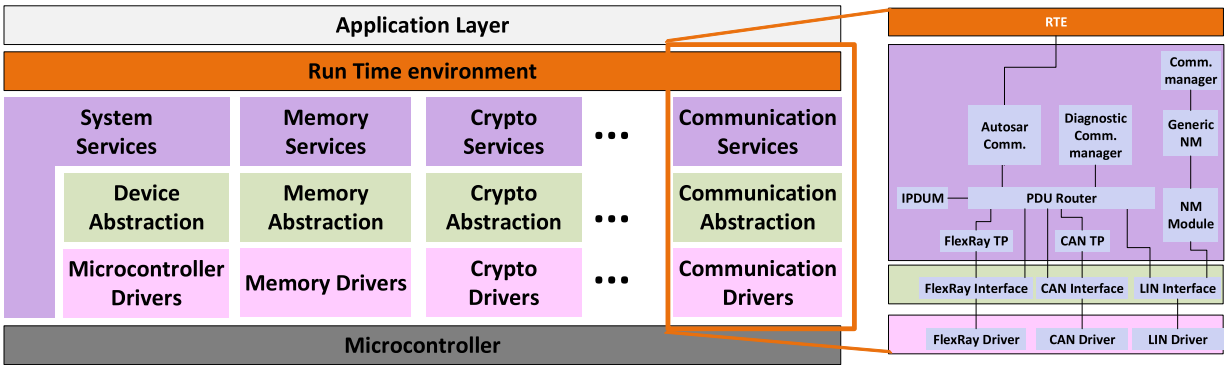

FIGURE 18. Simplified AUTOSAR stack (adapted from [68]).

this redundancy of the safety related communication over alternate paths.

Within the eGW, FRER can be integrated in the processing stage, as one more task in the stack which handles this generation and elimination of replicates, offloading the CPU from this reliability related processing. More details on how FRER algorithm can be deployed in the eGW architecture are described in [56].

IX. eGW SoC ARCHITECTURE AND AUTOSAR: HW VS SW CENTRICITY

In this section, we showcase how the eGW SoC architecture is compatible and integrable with the AUTOSAR Classic software stack. This compatibility is a key aspect towards a possible future adoption of eGW architecture in the industry, for instance in the way of new networking HW peripherals integrable in next-generation networking SoC devices and adopted in automotive and AUTOSAR standard. Furthermore, it is the main reason why it is possible to embed such a wide range of functionalities related to IVNs, and definitely the enabler of the integration of HW-Accelerated DDS. In order to show this concept, we start with an overview of AUTOSAR software stack and afterwards detail how each of the layers maps to eGW SoC architecture.

On the left side of Fig. 17 the AUTOSAR Classic SW stack is shown. From top to bottom, AUTOSAR defines an application layer where the different functionalities required in the vehicle organized in SW components are deployed. This application layer runs on top of the Run Time Environment (RTE). Below RTE there are different software layers that provide functionalities related to lower level infrastructure. On one side, there is a layer for service related functionalities which allows for supporting the service oriented architecture. This is key in providing flexibility towards distribution of services across the network supporting the SDN paradigm. On the other side, there are different HW abstraction layers allowing for making the higher layers independent of the lower layers. Particularly, this abstraction is divided between the MCU abstraction layer, which provides the required APIs to interact with the particular MCU in each implementation, and the ECU abstraction layer, which abstracts the whole ECU, where one or more MCUs may be in place. Additionally, there is a set of drivers that manage the communication between the RTE and the MCU, skipping the other layers when necessary.

On the right side of Fig. 17, we show how eGW maps to this software layers defined in AUTOSAR. On the top, the application layer runs on the eGW CPU, similarly to any other AUTOSAR implementation. There, the SW components responsible for running the applications of each automotive domain are deployed, i.e., ADAS/AD, body/comfort, cockpit/infotainment, powertrain/chassis and connectivity. Then, all the HW abstraction layers together with services and RTE are absorbed by the IPCores that compose eGW data-path and their interconnections. The ECU and microcontroller (MCU) abstraction layer are simplified in the CPU-HW interface

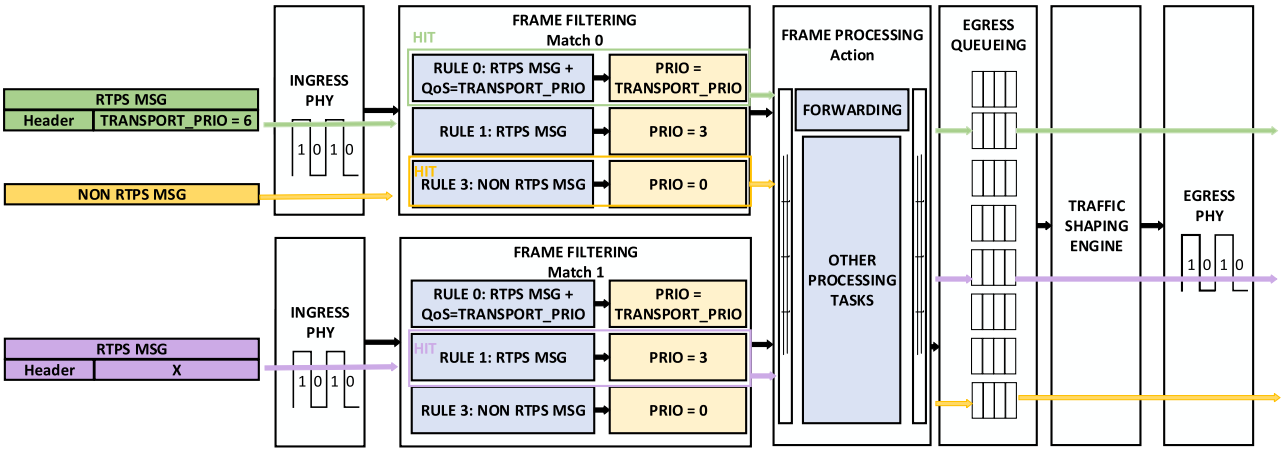

FIGURE 19. PoC — eGW as a switch between publishers and subscribers.

(yellow line in the figure), together with the memory map embedded in each of the IPCores. This simplification is possible thanks to the integration of functionalities directly in HW, which "lift up" the abstraction required by the CPU, covering most of the lower layers and reducing the complexity significantly.

The communication drivers are split across the different IPCores of eGW data-path. Fig. 18 shows a more detailed view of the functionalities embedded in these drivers in AUTOSAR specification. On one side, the drivers manage the communication with different network protocols (CAN, Ethernet, FlexRay, etc. in the figure). This functionality is absorbed in the Normalizer stage of eGW together with the SDN strategy previously described. Furthermore, eGW allows to integrate new network protocols by writing a different configuration in the Normalizer Memory Map. In AUTOSAR, the integration of a new protocol requires the the standardization and development of a new Basic Software (BSW) module or a custom Complex Device Driver (CDD). So, again, we see how eGW simplifies the management of complexity through HW integration. On the other side, the drivers perform also routing/diagnostics functionalities, that are split between Normalizer, Filtering and Action stage in eGW, providing all these functionalities in HW, without intervention of the CPU, increasing performance while reducing CPU load.

Finally, the Services Layer is absorbed between the Match and Action stages of eGW, by integrating the required IPCores for each service, together with the eGW data-path itself and the queuing and DAE strategies described above, as we have shown for DDS in the previous sections.

Overall, we see how eGW remains compatible with AUTOSAR and how it enables the integration of modern functionalities and services such as DDS in a simple way. More importantly, eGW is able to reduce the complexity of handling and orchestrating all the required functionalities while providing maximum performance, through a pioneer fully HW-centric approach.

X. PROOF OF CONCEPT

This section shows experimental results of some of the concepts defined above, synthesized on the FPGA of a Xilinx Zynq UltraScale+ ZU19EG SoC-based platform [69]. We evaluate the impact in performance of QoS both for the case when eGW is a network switch between publishers and subscribers, and when it is acting as a publisher or subscriber. For the PoC design, we use the previously described eGW Builder Tool in order to generate the corresponding design file for each test case. For the experimentation and data analysis we follow the approach described in [61]. Mainly, we base the experimentation on the use of standard PCAP files in order to inject/log traffic to/from the system. Afterwards, we process the collected PCAPs with Python scripts that allow to extract the data we are looking for.

A. TEST CASE 1: eGW AS A SWITCH BETWEEN PUBLISHERS AND SUBSCRIBERS

In this subsection, we evaluate the performance improvement in terms of QoS of RTPS messages provided by the eGW DDS support when eGW is a switch between publisher and subscribers, i.e. eGW is neither transmitter nor receiver of the RTPS messages. For this, we focus on the TRANSPORT_PRIORITY QoS policy which provides a simple yet powerful mechanism to control the priority with which a frame is transmitted through the network. To evaluate this feature, we run experiments with different configurations regarding the priority used for transportation of RTPS traffic. The experiments run are the following:

- **Experiment 1 — No prioritization of RTPS traffic**: First, we measure the delay observed for each frame without using any prioritization or traffic shaping mechanism, just a FIFO approach with all the traffic using the same queue inside the queueing modules.
- **Experiment 2 — Prioritization of RTPS traffic over Best Effort traffic**: Second, we define a rule in the filtering stage that recognises RTPS frames and assigns them a priority

higher than the rest of the traffic, which is considered best effort in the eGW.

- **Experiment 3 — Prioritization of RTPS traffic according to TRANSPORT-PRIORITY QoS**: Third, we add more rules in the filtering stage that not only recognise RTPS frames, but also inspect the QoS inline parameters. The priority used for the transmission of traffic in the egress stage of eGW is determined by the content of the TRANSPORT-PRIORITY QoS policy.

The PoC runs on a GW platform based on eGW architecture, particularized for this use case as depicted in Fig. 19. In this case, we instantiate two ingress ports and one egress port, with Ethernet connections of 100 Mbps. We inject the same traffic (at the same time) in the two ingress ports, and forward all the traffic to the egress port. The traffic in each of the ports consists of 1000 frames of 64 Bytes which are sent every 200 μs. We define 8 different traffic classes, following the IEEE TSN standardization approach [40], where Traffic Class 7 represents the highest priority, and Traffic Class 0 represents the lowest priority. As shown in Fig. 19, we define three filtering rules to differentiate RTPS messages with TRANSPORT_PRIORITY QoS policy, RTPS messages in general with other QoS properties and non-RTPS messages. For each of them, a different priority is defined and used internally in eGW. This way, frames are stored in different queues of the egress stage (based on availability and not on priority, since priority is embedded in the instruction frame). Finally, the TSE is able to select between the frames based on the priority field of the instruction frame, closing thus the loop and allowing to use the QoS properties defined by the DDS middleware. However, as detailed before, not all the experiments are sensitive to this traffic classification. In the first experiment there is no prioritization of traffic (i.e. all frames fall under Rule 3 in Fig. 19). In the second one, all RTPS messages are given the same priority (Rules 2 and 3 apply, differentiating RTPS from non RTPS traffic). Finally, on the third experiment, the TRANSPORT-PRIORITY QoS policy is used (Rules 1, 2 and 3 apply, differentiating RTPS messages with TRANSPORT_PRIORITY QoS, RTPS messages with other QoS properties and non RTPS messages). In our experiments, the TRANSPORT_PRIORITY used in the messages corresponds to the traffic class of the frames, for the sake of easing the final data analysis. We generate the traffic based on PCAP files with CANoe tool from Vector [70].

The detailed results for each of the experiments are shown in Table 11. For each of the experiments, we show the maximum, minimum and average delay for the flows of each traffic class. Traffic Class 7 represents traffic with highest priority, and Traffic Class 0 represents traffic with lowest priority. As we can see in the tables, in the case of Experiment 1, eGW does not differentiate among traffic classes and therefore the average delay for all classes is the same. When we introduce the prioritization strategy, we already see some improvements (Experiment 2) regarding the delay of the RTPS traffic, which further improves when introducing

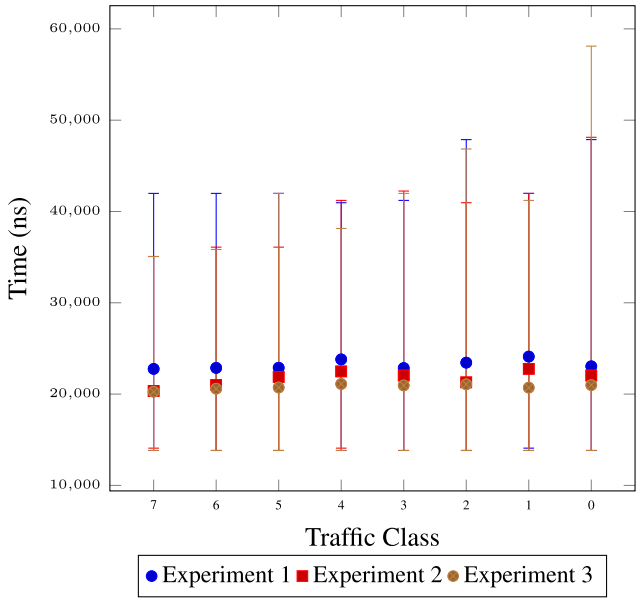

FIGURE 20. PoC — plot of delay measurement of frames across experiments.

the use of TRANSPORT-PRIORITY QoS policy (Experiment 3). The rows highlighted in bold in Table 11 show the delay experienced by frames corresponding to the highest priorities (Traffic Class 6 and 7). There we see how in the first experiment these flows have similar delay to others, while in experiments 2 and 3 they benefit from higher transport priority. On the other side, the worst case delay for traffic with lower priority increases significantly in exchange for reducing the worst case delay of high priority traffic, as it could be expected. This effect appears because the experiments are using two ingress ports at maximum rate to transmit traffic over one single egress port, taking the system to the limit. Therefore, the egress port needs to queue some of the low priority frames for a longer time in order to infer the lowest possible delay over new coming high priority traffic. In Table 12, we compare a summary of the results focusing on the average delay per traffic class in each of the approaches. We see that, in general, the average delay of all traffic classes improves, with higher improvements related to higher traffic prioritization (improvement for Traffic Class 6 and 7 highlighted in bold).

Finally, Fig. 20 provides a graphical view of the results. Again, we see the results for each of the traffic classes, being TC 7 and 6 the ones with highest priority. We represent the range of delay from minimum to maximum within each traffic class with vertical lines, and the average with dots. The different experiments are differentiated with colors and superposed in the figure to ease visual comparison. Experiment one is plotted in blue, experiment 2 in red and experiment 3 in brown. The Figure shows how the introduction of prioritization improves the average response time of all traffic classes for this particular use case. Furthermore, we also see how the use of specific priorities for each traffic class reduces the

TABLE 11. PoC – measured frames delay (in ns) across the different experiments.

Experiment Configuration		Experiment 1 Delay			Experiment 2 Delay			Experiment 3 Delay		
Traffic Class	Frames in	Minimum	Maximum	Average	Minimum	Maximum	Average	Minimum	Maximum	Average
7	**125**	**13824**	**41984**	**22759**	**14080**	**35072**	**20342**	**13824**	**35072**	**20256**
6	**125**	**13824**	**41984**	**22870**	**13824**	**36096**	**21004**	**13824**	**35840**	**20592**
5	125	13824	41984	22884	13824	36096	21880	13824	41984	20721
4	125	13824	40960	23799	14080	41216	22484	13824	38144	21110
3	125	13824	41216	22851	13824	42240	22032	13824	41984	20948
2	125	13824	47872	23443	13824	40960	21288	13824	46848	21071
1	125	14080	41984	24107	13824	41984	22769	13824	41216	20707
0	1125	13824	47872	23042	13824	48128	22034	13824	58112	20985

*Traffic Classes 6 and 7 represent highest priority traffic

TABLE 12. PoC – average delay comparison across the different experiments (total measurements in ns).

#	TC 7	TC 6	TC 5	TC 4	TC 3	TC 2	TC 1	TC 0
1	22759 (Ref)	22870 (Ref)	22884 (Ref)	23799 (Ref)	22851 (Ref)	23443 (Ref)	24107 (Ref)	23042 (Ref)
2	**20342 (-10,6%)**	**21004 (-8,1%)**	21880 (-4,4%)	22484 (-5,5%)	22032 (-3,6%)	21288 (-9,2%)	22769 (-5,5%)	22034 (-4,4%)
3	**20256 (-11%)**	**20592 (-10%)**	20721 (-9,5%)	21110 (-11,3%)	20948 (-8,3%)	21071 (-9,2%)	20707 (-14,1%)	20985 (-4,37%)

#: Experiment Number
*Traffic Classes 6 and 7 represent highest priority traffic

delay of higher priority classes, allowing to guarantee a maximum worst case for a given traffic class. We see how the introduction of priorities per traffic class reduces the maximum delay of high priority traffic (TC 6 and 7) in exchange for longer worst case delays in the lower priority traffic classes. The big variations observed in the experiments are caused by the traffic pattern used, where two ports are sending a burst of traffic to one single egress port simultaneously. During the experiments, we saw that the traffic pattern used has a great impact on the result. One shortcoming we identified, is that there is currently no available and standardized data set that can be used to simulate vehicular traffic for experiments. Authors in [71] identify this issue too, and provide some guidelines on how to build the traffic that can be used in IVN experiments. In our case, we follow these guidelines and randomize the generation of frames when possible in order to get the most realistic results we can. From our perspective, this is an opportunity for future research as well, which could represent an important contribution in the field.

B. TEST CASE 2: eGW AS A PUBLISHER OR SUBSCRIBER

In this subsection, we evaluate some of the benefits of performing certain DDS features in a HW accelerator when eGW is acting as publisher or subscriber. In particular, we evaluate the benefit of offloading a simple task such as the LIVELINESS QoS policy. As described before, this policy mainly defines the periodicity of Heartbeat messages that need to be generated and transmitted by writers. The benefits of offloading such functionality are two-fold:

- **CPU-load reduction**: On one side, the CPU is free of this task and can dedicate its resources to other processing tasks that require software capabilities (e.g. due to algorithm complexity or high silicon cost of a HW alternative). This is a qualitative benefit that is difficult to measure since it depends on the applications running on the CPU.
- **Better timing accuracy**: On the other side, the time precision achieved by the HW implementation is typically higher (i.e. the jitter observed regarding the established period is expected to be smaller). This aspect can be quantitatively evaluated, so we focus on this second benefit in this PoC.

In order to evaluate the difference in the jitter between messages at receiver side, we deploy two implementations of the LIVELINESS QoS policy: a software-based implementation and a HW-accelerated implementation. On the software-based implementation, we define a function that sends frames with a certain periodicity by writing the corresponding registers of the HW driver. No other functionalities are present in the CPU at run-time and no other traffic is present in the system. Although being a simplistic system implementation, it allows us to compare the impact of the SW implementation running on the CPU Operating System versus the HW-accelerated functionality. On the HW-based implementation we implement the publisher support block described in Fig. 16. Both the SW and HW implementations are configured to generate alive messages every second, and we run the experiment on both platforms for 1000 seconds. The results are summarized in Table 13 and Fig. 21. As expected, the jitter of the HW implementation (250 ns) is much smaller than the jitter experienced by the SW implementation (>1ms). Considering that the SW implementation used for this experimentation is a very simple one without interferences of other tasks/threads in parallel, it is reasonable to foresee that the benefit would be even higher on an ECU loaded with many different applications running concurrently. It is important to note that we use the timestamp provided by the CANoe tool [70] when recording the resulting PCAP files. This means that measurements include delay

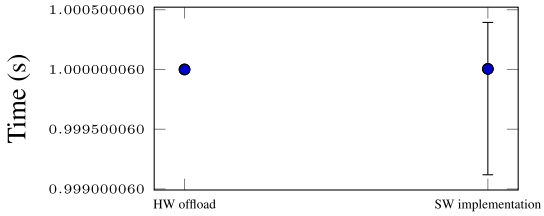

FIGURE 21. PoC – plot of jitter measurement of Heartbeat messages.

TABLE 13. PoC – results of jitter measurement of Heartbeat messages.

Statistic	SW Implementation	HW offload
Maximum time (s)	1.000393828	1.000000250
Average time (s)	1.000004977	1.000000147
Minimum time (s)	0.999118813	1
Jitter (Max-Min) (s)	0.001275015	0.000000250

and jitter related to the tool itself, and not only to the eGW implementation. However, since this is the same for both the SW and HW implementation this is just an offset in both cases and does not influence the purpose of the comparison.

XI. CONCLUSION

In this paper, we have illustrated the rationale behind the transition of the automotive market towards service-oriented architectures (SoA) and we have illustrated how some use cases can take advantage of the functionalities provided by modern middlewares.

The research has then contributed in bringing the DDS technology to the next-level, by proposing the transition of some DDS functionalities from software to hardware accelerators. This approach allows to either avoid the need of powerful and expensive MCUs/CPUs to execute the DDS middleware and improve the predictability of the overall system. The concept of this transition has been not only detailed but also deployed through a Proof of Concept. For such a goal, the eGW SoC architecture has been developed showcasing the possibility to manage DDS requirements at HW level through the right management of queueing, arbitration and scheduling of frames with the different proposed IPCores. With this, the suitability of the HW-centric approach for high-performance DDS deployment is successfully demonstrated. Moreover, we have shown some insights about the reasonable easiness of porting the HW-centric DDS solution to be adopted and standardized in AUTOSAR by means of new standardizable HW peripherals integrable in future next-generation automotive-related SoC devices as accelerators.

The work done proves also that the evaluated DDS QoS policies can coexist in a compatible manner with other TSN standards and IVN functional safety mechanisms required in next-generation autonomous-connected-electric-and-shared (ACES) vehicles. The work describes also the compatibility of DDS with part of TSN P802.1DG and Functional Safety ISO 26262 standards for certain automotive in-vehicle networking uses cases.

All in all, this work pioneers the deployment of HW-centric DDS in cyber-physical systems, particularized here in a zonal gateway networking SoC device for automotive-related scenarios. The work shows how the presented eGW SoC architecture enables not only the integration of HW-centric DDS and TSN features but also the compatibility with AUTOSAR software stack. The research has also contributed to bridge the gap in automotive software complexity thanks to a pioneer HW-oriented approach that integrates SDN, TSN and DDS.

REFERENCES

[1] *Road Vehicles—Functional Safety—Part 1: Vocabulary*, Standard ISO 26262-1:2018, 2018. [Online]. Available: https://www.iso.org/standard/68383.html
[2] AUTOSAR. *Classic Platform*. Accessed: Sep. 15, 2022. [Online]. Available: https://www.autosar.org/standards/classic-platform/
[3] *Standardized E-Gas Monitoring Concept for Gasoline and Diesel Engine Control Units Version 6.0*, EGAS Workgroup, Germany, 2015.
[4] *OSEK/VDX Operating System Specification 2.2.3*, OSEK, Germany, Feb. 2005.
[5] McKinsey. *The Case for an End-to-End Automotive Software Platform*. [Online]. Available: https://www.mckinsey.com/industries/automotive-and-assembly/our-insights/the-case-for-an-end-to-end-automotive-software-platform
[6] Luc van Dijk. (2017). *Future Vehicle Networks and ECUS Architecture and Technology Considerations*. NXP Semiconductors. [Online]. Available: https://www.nxp.com/docs/en/white-paper/FVNECUA4WP.pdf
[7] H. Askaripoor, M. H. Farzaneh, and A. Knoll, "E/E architecture synthesis: Challenges and technologies," *Electronics*, vol. 11, no. 4, p. 518, Feb. 2022.
[8] K. Strandberg, T. Olovsson, and E. Jonsson, "Securing the connected car: A security-enhancement methodology," *IEEE Veh. Technol. Mag.*, vol. 13, no. 1, pp. 56–65, Mar. 2018.
[9] AUTOSAR. *Adaptive Platform*. Accessed: Sep. 15, 2022. [Online]. Available: https://www.autosar.org/standards/adaptive-platform/
[10] E. W. Dijkstra, "On the role of scientific thought," in *Selected Writings on Computing: A Personal Perspective*. Cham, Switzerland: Springer, 1982, pp. 60–66.
[11] S. Tuohy, M. Glavin, C. Hughes, E. Jones, M. Trivedi, and L. Kilmartin, "Intra-vehicle networks: A review," *IEEE Trans. Intell. Transp. Syst.*, vol. 16, no. 2, pp. 534–545, Apr. 2014.
[12] O. Alparslan, S. Arakawa, and M. Murata, "Next generation intra-vehicle backbone network architectures," in *Proc. IEEE 22nd Int. Conf. High Perform. Switching Routing (HPSR)*, Jun. 2021, pp. 1–7.
[13] J. Walrand, M. Turner, and R. Myers, "An architecture for in-vehicle networks," *IEEE Trans. Veh. Technol.*, vol. 70, no. 7, pp. 6335–6342, Jul. 2021.
[14] G. Patti, L. Lo Bello, and L. Leonardi, "Deadline-aware online scheduling of TSN flows for automotive applications," *IEEE Trans. Ind. Informat.*, early access, Jun. 17, 2022, doi: 10.1109/TII.2022.3184069.
[15] Y. Seol, D. Hyeon, J. Min, M. Kim, and J. Paek, "Timely survey of time-sensitive networking: Past and future directions," *IEEE Access*, vol. 9, pp. 142506–142527, 2021.
[16] OMG. *Data Distribution Service (DDS) Version 1.4*. Accessed: Sep. 15, 2022. [Online]. Available: https://www.omg.org/spec/DDS/
[17] *Robot Operating System (ROS)*. Accessed: Sep. 15, 2022. [Online]. Available: https://www.ros.org
[18] Apex.AI, Inc. *Apex.OS*. Accessed: Sep. 15, 2022. [Online]. Available: https://www.apex.ai
[19] M. Pohnl, A. Tamisier, and T. Blass, "A middleware journey from micro-controllers to microprocessors," in *Proc. Design, Autom. Test Eur. Conf. Exhib. (DATE)*, Mar. 2022, pp. 282–286.
[20] AUTOSAR. (2022). *13th AUTOSAR Open Conference (AOC)*. [Online]. Available: https://www.autosar.org/news-events/aoc2022/
[21] *Object Management Group*. Accessed: Sep. 15, 2022. [Online]. Available: https://www.omg.org
[22] OMG. *The Real-Time Publish-Subscribe Protocol DDS Interoperability Wire Protocol (DDSI-RTPSTM) Specification Version 2.5*. [Online]. Available: https://www.omg.org/spec/DDSI-RTPS
[23] OMG. *DDS-TSN Request For Proposals*. Accessed: Sep. 15, 2022. [Online]. Available: https://www.omg.org/news/releases/pr2018/10-08-18.htm

[24] AUTOSAR. *SOME/IP Protocol Specification*. Accessed: Sep. 15, 2022. [Online]. Available: https://www.autosar.org/fileadmin/user_upload/standards/foundation/21-11/AUTOSAR_PRS_SOMEIPProtocol.pdf
[25] H. Askaripoor, M. H. Farzaneh, and A. Knoll, "E/E architecture synthesis: Challenges and technologies," *Electronics*, vol. 11, no. 4, p. 518, Feb. 2022.
[26] A. G. Marino, F. Fons, and J. M. M. Arostegui, "The future roadmap of in-vehicle network processing: A HW-centric (R-)evolution," *IEEE Access*, vol. 10, pp. 69223–69249, 2022.
[27] A. A. Kane, A. G. Marino, F. Fons, S. Nueesch, P. Serwa, and M. Schoetz, "Elastic gateway functional safety architecture and deployment: A case study," *IEEE Access*, vol. 10, pp. 91771–91801, 2022.
[28] F. Fons and M. Fons, "FPGA-based automotive ECU addresses AUTOSAR and ISO 26262 standards," *Xcellence Automot. Appl.*, vol. 1, pp. 20–31, 1st Quart., 2012.
[29] F. Fons, M. Fons, P. Olivier, and A. Weimerskirch, "A modular, reconfigurable and updateable embedded cyber security hardware solution for automotive," in *Proc. Embedded World Conf.*, 2017.
[30] S. Shreejith and S. A. Fahmy, "Security aware network controllers for next generation automotive embedded systems," in *Proc. 52nd Annu. Design Autom. Conf.*, Jun. 2015, pp. 1–6.
[31] P. Bellavista, A. Corradi, L. Foschini, and A. Pernafini, "Data distribution service (DDS): A performance comparison of OpenSplice and RTI implementations," in *Proc. IEEE Symp. Comput. Commun. (ISCC)*, Jul. 2013, pp. 377–383.
[32] T. Wu, B. Wu, S. Wang, L. Liu, S. Liu, Y. Bao, and W. Shi, "Oops! It's too late. Your autonomous driving system needs a faster middleware," *IEEE Robot. Autom. Lett.*, vol. 6, no. 4, pp. 7301–7308, Oct. 2021.
[33] S. Profanter, A. Tekat, K. Dorofeev, M. Rickert, and A. Knoll, "OPC UA versus ROS, DDS, and MQTT: Performance evaluation of Industry 4.0 protocols," in *Proc. IEEE Int. Conf. Ind. Technol. (ICIT)*, Feb. 2019, pp. 955–962.
[34] OpenADX. *iceoryx—True Zero-Copy Inter-Process-Communication*. Accessed: Sep. 15, 2022. [Online]. Available: https://github.com/eclipse-iceoryx/iceoryx
[35] Y. Chen and T. Kunz, "Performance evaluation of IoT protocols under a constrained wireless access network," in *Proc. Int. Conf. Sel. Topics Mobile Wireless Netw. (MoWNeT)*, Apr. 2016, pp. 1–7.
[36] *MQTT*, OASIS, Manchester, U.K., 2014.
[37] Vector, "Middleware protocols in the automobile: Service-oriented, data-centric or RESTful?" *Elektronik Automot. Mag.*, vol. 3, pp. 18–21, Mar. 2020.
[38] A. Ioana, A. Korodi, and I. Silea, "Automotive IoT Ethernet-based communication technologies applied in a V2X context via a multi-protocol gateway," *Sensors*, vol. 22, no. 17, p. 6382, Aug. 2022.
[39] C. Menard, A. Goens, M. Lohstroh, and J. Castrillon, "Achieving determinism in adaptive AUTOSAR," in *Proc. Design, Autom. Test Eur. Conf. Exhib. (DATE)*, Mar. 2020, pp. 822–827.
[40] IEEE. *Time Sensitive Networking Working Group*. Accessed: Sep. 15, 2022. [Online]. Available: https://1.ieee802.org/tsn/
[41] A. Nasrallah, S. A. Thyagaturu, Z. Alharbi, C. Wang, X. Shao, M. Reisslein, and H. ElBakoury, "Ultra-low latency (ULL) networks: The IEEE TSN and IETF DetNet standards and related 5G ULL research," *IEEE Commun. Surveys Tuts.*, vol. 21, no. 1, pp. 88–145, 1st Quart., 2019.
[42] L. Lo Bello, G. Patti, and G. Vasta, "Assessments of real-time communications over TSN automotive networks," *Electronics*, vol. 10, no. 5, p. 556, Feb. 2021.
[43] S. B. H. Said, Q. H. Truong, and M. Boc, "SDN-based configuration solution for IEEE 802.1 time sensitive networking (TSN)," *ACM SIGBED Rev.*, vol. 16, no. 1, pp. 27–32, Feb. 2019.
[44] *Draft Standard for Local and Metropolitan Area Networks—Time-Sensitive Networking Profile for Automotive in-Vehicle Ethernet Communications*, Standard IEEE P802.1DG/D1.4, 2021.
[45] *IEEE Standard for Local and Metropolitan Area Networks—Timing and Synchronization for Time-Sensitive Applications*, IEEE Standard 802.1AS-2020, 2020.
[46] *IEEE Standard for Local and Metropolitan Area Networks—Bridges and Bridged Networks—Amendment 28: Per-Stream Filtering and Policing*, IEEE Standard 802.1Qci-2017, IEEE Standard 802.1Q-2014, IEEE Standard 802.1Qca-2015, IEEE Standard 802.1Qcd-2015, IEEE Standard 802.1Q-2014/Cor 1-2015, IEEE Standard 802.1Qbv-2015, IEEE Standard 802.1Qbu-2016, and IEEE Standard 802.1Qbz-2016, 2017, pp. 1–65.
[47] LAN Man, Standards Committee, and IEEE Computer, *IEEE Standard for Local and Metropolitan Area Networks—Frame Replication and Elimination for Reliability*, Standard IEEE802.1CB, 2017.
[48] *IEEE Standard for Local and Metropolitan Area Networks—Virtual Bridged Local Area Networks Amendment 12 : Forwarding and Queuing Enhancements for Time-Sensitive Streams*, Standard 802.1Qav-2009, 2010.
[49] *IEEE Standard for Local and Metropolitan Area Networks—Bridges and Bridged Networks—Amendment 25: Enhancements for Scheduled Traffic*, IEEE Standard 802.1Qbv-2015, IEEE Standard 802.1Q-2014, IEEE Standard 802.1Qca-2015, IEEE Standard 802.1Qcd-2015, and IEEE Standard 802.1Q-2014/Cor 1-2015, 2016, pp. 1–57.
[50] *IEEE Standard for Local and Metropolitan Area Networks—Bridges and Bridged Networks—Amendment 29: Cyclic Queuing and Forwarding*, IEEE Standard 802.1Qch-2017, IEEE Standard 802.1Q-2014, IEEE Standard 802.1Qca-2015, IEEE Standard 802.1Qcd(TM)-2015, IEEE Standard 802.1Q-2014/Cor 1-2015, IEEE Standard 802.1Qbv-2015, IEEE Standard 802.1Qbu-2016, IEEE Standard 802.1Qbz-2016, and IEEE Standard 802.1Qci-2017, 2017, pp. 1–30.
[51] *IEEE Standard for Local and Metropolitan Area Networks—Bridges and Bridged Networks—Amendment 34: Asynchronous Traffic Shaping*, IEEE Standard 802.1Qcr-2020, IEEE Standard 802.1Q-2018, IEEE Standard 802.1Qcp-2018, IEEE Standard 802.1Qcc-2018, IEEE Standard 802.1Qcy-2019, and IEEE Standard 802.1Qcx-2020, 2020, pp. 1–151.
[52] *IEEE Standard for Local and Metropolitan Area Networks—Bridges and Bridged Networks—Amendment 26: Frame Preemption*, IEEE Standard 802.1Qbu-2016, IEEE Standard 802.1Q-2014, 2016, pp. 1–52.
[53] A. G. Mariño, F. Fons, L. Ming, and J. M. M. Arostegui, "PDU normalizer engine for heterogeneous in-vehicle networks in automotive gateways," in *Applied Reconfigurable Computing. Architectures, Tools, and Applications*, S. Derrien, F. Hannig, P. C. Diniz, and D. Chillet, Eds. Cham, Switzerland: Springer, 2021, pp. 140–155.
[54] *Information Processing Systems—Open Systems Interconnection—Basic Reference Model*, Standard ISO 7498-2:1989, 1989. [Online]. Available: https://www.iso.org/standard/14256.html
[55] A. G. Marino, F. Fons, A. Gharba, L. Ming, and J. M. M. Arostegui, "Elastic queueing engine for time sensitive networking," in *Proc. IEEE 93rd Veh. Technol. Conf. (VTC-Spring)*, Apr. 2021, pp. 1–7.
[56] A. G. Mariño, A. A. Kane, F. Fons, and J. M. M. Arostegui, "Enhancements for hardware-based IEEE802.1CB embedded in automotive gateway system-on-chip," in *Proc. Symp. Architectures Netw. Commun. Syst.*, New York, NY, USA, Dec. 2021, pp. 31–37.
[57] A. G. Mariño, F. Fons, Z. Haigang, and J. M. M. Arostegui, "Loopback strategy for in-vehicle network processing in automotive gateway network on chip," in *Proc. 14th Int. Workshop Netw. Chip Architectures*, New York, NY, USA, Oct. 2021, pp. 22–28.
[58] A. G. Marino, F. Fons, Z. Haigang, and J. M. M. Arostegui, "Traffic shaping engine for time sensitive networking integration within in-vehicle networks," in *Proc. IEEE Veh. Netw. Conf. (VNC)*, Nov. 2021, pp. 182–189.
[59] A. G. Marino, F. Fons, Z. Haigang, and J. M. M. Arostegui, "Loopback strategy for TSN-compliant traffic queueing and shaping in automotive gateways," in *Proc. IEEE Conf. Netw. Function Virtualization Softw. Defined Netw. (NFV-SDN)*, Nov. 2021, pp. 47–53.
[60] A. G. Marino, N. H. Naganath, F. Fons, and J. M. M. Arostegui, "Build automation framework for architecture design of automotive elastic gateway," in *Proc. Embedded World Conf.*, 2022, pp. 728–742.
[61] A. G. Marino, N. N. Halinge, F. Fons, and J. M. M. Arostegui, "Build automation framework for design validation of automotive gateway controllers," in *Proc. IFIP Netw. Conf. (IFIP Netw.)*, Jun. 2022, pp. 1–6.
[62] OMG. *DDS for eXtremely Resource Constrained Environments Version 1.0*. Accessed: Sep. 15, 2022. [Online]. Available: https://www.omg.org/spec/DDS-XRCE
[63] X. Zhang, J. Wang, C. Zhu, Y. Lin, J. Xiong, W.-M. Hwu, and D. Chen, "DNNBuilder: An automated tool for building high-performance DNN hardware accelerators for FPGAs," in *Proc. Int. Conf. Comput.-Aided Design*, Nov. 2018, pp. 1–8.
[64] J. Yan, W. Quan, X. Yang, W. Fu, Y. Jiang, H. Yang, and Z. Sun, "TSN-builder: Enabling rapid customization of resource-efficient switches for time-sensitive networking," in *Proc. 57th ACM/IEEE Design Autom. Conf. (DAC)*, Jul. 2020, pp. 1–6.
[65] T. Agarwal, P. Niknejad, M. R. Barzegaran, and L. Vanfretti, "Multi-level time-sensitive networking (TSN) using the data distribution services (DDS) for synchronized three-phase measurement data transfer," *IEEE Access*, vol. 7, pp. 131407–131417, 2019.
[66] *Using DDS Over TSN to Support NATO Generic Vehicle Architecture (NGVA) for Land Systems*, RELYUM Real Time Innov. (RTI), Erandio, Spain, 2019.

[67] S. Sudhakaran, V. Mageshkumar, A. Baxi, and D. Cavalcanti, "Enabling QoS for collaborative robotics applications with wireless TSN," in *Proc. IEEE Int. Conf. Commun. Workshops (ICC Workshops)*, Jun. 2021, pp. 1–6.
[68] AUTOSAR. *AUTOSAR Layered Software Architecture*. Accessed: Sep. 15, 2022. [Online]. Available: https://www.autosar.org/fileadmin/user_upload/standards/classic/21-11/AUTOSAR_EXP_LayeredSoftwareArchitecture.pdf
[69] *ProFPGA Zynq UltraScale+ ZU19EG*. Accessed: Sep. 15, 2022. [Online]. Available: https://www.profpga.com/products/fpga-modules-overview/zynq-ultrascale-based/profpga-zu19eg
[70] Vector. *Canoe*. Accessed: Sep. 15, 2022. [Online]. Available: https://www.vector.com/int/en/products/products-a-z/software/canoe/
[71] F. Rezabek, M. Bosk, T. Paul, K. Holzinger, S. Gallenmüller, A. Gonzalez, A. Kane, F. Fons, Z. Haigang, G. Carle, and J. Ott, "EnGINE: Flexible research infrastructure for reliable and scalable time sensitive networks," *J. Netw. Syst. Manage.*, vol. 30, no. 4, Oct. 2022, Art. no. 74, 10.1007/s10922-022-09686-0.

CLAUDIO SCORDINO (Member, IEEE) received the M.Sc. degree in computer engineering and the Ph.D. degree in computer science from the University of Pisa, in 2003 and 2007, respectively. He collaborated with Scuola Superiore Sant'Anna and the University of Pittsburgh about research on power-aware real-time operating systems. He has collaborated with the Linux Kernel Community, especially for the development of the SCHED_DEADLINE CPU scheduler. His research interests include real-time operating systems, middle wares, and hypervisors. He is an AUTOSAR Member, actively collaborating to the standard.

ANGELA GONZALEZ MARIÑO received the bachelor's degree in telecommunications engineering from the Universidade de Vigo (UVIGO), Vigo, Spain, in 2015, and the master's degree in electronics engineering systems from the Universidad Politecnica de Madrid (UPM), Madrid, Spain, in 2016. She is currently pursuing the Ph.D. degree with the Universitat Politècnica de Catalunya (UPC), Barcelona, Spain.

She was a Research and Development Electronics Engineer at HP Inc., Barcelona, from 2016 to 2020, designing electronics for large format printers and supporting the full product lifecycle development. She is currently with the Automotive Engineering Laboratory of Munich Research Center, Huawei Technologies, Munich, Germany, focusing on HW accelerators design for automotive networking solutions. Her current research interests include HW design for automotive in-vehicle networks and system on chip design.

FRANCESC FONS (Senior Member, IEEE) received the bachelor's degree in electrical engineering, the master's degree in automatic control and industrial electronics engineering, and the Ph.D. degree in electronics technology from the Universitat Rovira i Virgili (URV), Tarragona, Spain, in 1995, 2001, and 2012, respectively.

He has focused his professional career on the automotive electronics industry, working on Research and Development in the areas of embedded software, systems, hardware, and networks. Along his career, he has been with different automotive Tier 1 and Tier 2 suppliers from USA, Germany, and China, and has participated in the successful launch of many commercial products for OEMs in Europe and Asia. Currently, he is with the Automotive Engineering Laboratory of Munich Research Center, Huawei Technologies.