A new design for data-centric Ethernet communication with tight synchronization requirements for automated vehicles

Kai-Björn Gemlau, Jonas Peeck, Nora Sperling, Phil Hertha, Rolf Ernst
Institute of Computer and Network Engineering, TU Braunschweig
Braunschweig, Germany
gemlau|peeck|noras|philh|ernst@ida.ing.tu-bs.de

***Abstract*—Deterministic communication is a key challenge for modern embedded real-time systems. This is especially true for the automotive domain, where safety-critical functions are distributed over multiple electronic control units (ECUs) across the vehicle. To handle the increased complexity together with the higher data volume, Ethernet is considered as a universal media. Consequently, a great effort has been spent on making Ethernet predictable to satisfy timing and safety requirements. However, the communication stacks (COM-stacks) that build the bridge between applications and the network did not receive the same attention. Until now it has been dealt with as legacy software and overloaded with additional features to support the new multi-level protocols. Moreover, with the shift to automated vehicles, communication requirements will change fundamentally, transforming the car to a data centric system.**

In this paper we highlight the communication requirements of future autonomous cars and show why today's COM-stack designs are not suited to meet them. We present a novel design approach that ensures predictable timing and resource sharing while focusing on a minimalist and modular design. We also show how it fits to a state-of-the-art middleware, while keeping the complexity as small as possible.

***Index Terms*—Distributed real-time systems, timing analysis, data flow, cause-effect chains, logical execution time**

I. Introduction

Ethernet based communication is omnipresent in modern embedded system platforms. Thus, Ethernet was also introduced in the safety-critical automotive domain to benefit from the high bandwidth and a standardized topology, that obviates complex gateways between different types of buses like CAN, LIN or FlexRay. From the software perspective, the unified network access for all ECUs over a standardized middleware implementation is a strategy that simplifies the design process. While Ethernet originally was designed for throughput and scalability on a best effort basis, it brings up new challenges in case of safety-critical requirements like timing behavior or reliability, which are characteristic for the automotive domain.

In contrast to automotive busses like CAN or FlexRay, Ethernet has very limited support for isolation mechanisms regarding timing or buffers. This implies, that traffic streams from different criticality levels may interfere each other on a temporal or spatial level on shared resources like COM-stacks, network links or switches. An example would be highly critical steering control traffic and less critical diagnostic data that both share the same link. For this situation, the automotive safety standard ISO26262 claims a "sufficient level of independence" [1], such that the interference from the low critical on the high critical function is provable bounded.

Focused on the network itself, time sensitive networks (TSN) introduce scheduling extensions for Ethernet, like the time-aware shaper [2], to provide temporal isolation as well as low latencies, but their use is limited since the concept does not scale well for many competing traffic streams [3]. Moreover, this represents only a part of the overall communication architecture. In the following we will show, that COM-stacks as well as the middleware have to be considered too. One approach is to control the input of the network by a centralized unit, so that from a global point of view guarantees can be given to Ethernet links [4].

To identify the domain specific problems of automotive software, we have to take a look at the design process. Typically, MATLAB/Simulink [5] is used to model the periodic behavior of a control loop that meets a safety goal (e.g. lane keeping). This model can be enriched with constant delays to reflect execution times and communication delays. With distributed control applications, the overall dead time of the controller strongly depends on the communication overhead between the ECUs. Since Simulink assumes a constant delay, jitter must be avoided on the whole cause-effect chain. Otherwise the implementation does not reflect the model [6].

As previously proposed, one approach to this problem is the concept of system-level LET (SL LET) [6], which is an extension of the logical execution time (LET) paradigm [7]. The classical LET approach is focused on intra-ECU communication between task's, reading data at activation time and writing data after their LET, which is a fixed time greater than the tasks worst-case response time (WCRT). A cause-effect chain consisting of LET tasks has a deterministic end-to-end (E2E) latency with zero jitter, at the cost of always assuming the worst-case delay. SL LET extends this concept to the network using a specific *interconnect task*. It starts with the send operation in the COM-stack on the source ECU and terminates after the receive operation of the COM-stack on the destination ECU. Due to the poor possibilities of temporal and spatial isolation in the current Ethernet setup, the worst-case timing of a critical interconnect task will be

978-1-7281-4878-6/19/$31.00 ©2019 IEEE

very pessimistic. This motivates the first intention of the paper, the exploration of requirements and definition of design guidelines for a COM-stack which supports temporal isolation and significantly reduces the E2E delay for SL LET systems.

Now that a lot of effort has been spent on guarantees for Ethernet communication times of different criticalities ([3][6]), the requirements change fundamentally due to the goal of autonomous driving. While today's time-critical data is mainly limited to relatively small control data, extensive sensor data will also become time-critical when driving autonomously, since the loss of the human driver as a safety instance must be compensated.

Autonomous driving comprises long sensor to actor cause-effect chains, spreading over different ECUs, which are now also latency critical. The chains contain various data intensive tasks like sensor fusion, environment perception and planning. The underlying communication paths are managed by the middleware and processed by the COM-stack. Since the communication still has to meet an E2E deadline, we argue that there is a shift from *packet-oriented transmission* towards *object-oriented transmission*, where latency guarantees are assigned to a whole set of packets representing the object to transport (e.g. a picture). Moreover, the loss of even one Ethernet packet is often not acceptable in an object-oriented transmission, since then the whole object might become invalid or degraded. A classical transmission of a large object will produce a packet burst, stressing the buffers and other channels on the link, while using fragmentation on IP level, which is protected with complex protocols, such as TCP. As we can see, applying the current approach to object-oriented transmission will increase worst-case latency, leading to high pessimism in the system. This makes approaches like SL LET less beneficial, at least on the level of packets. Moreover, the latency of the jitter free SL LET approach will be even higher, since the LET will be based on the worst-case burst transmission latency. It is noteworthy that SOME/IP, as a relevant automotive middleware specification, claims to enable fast and efficient object-oriented transmission by SOME/IP-TP. [8]. However, this approach only splits objects with UDP (avoiding IP fragmentation), while leaving other requirements such as retransmission on packet loss unaddressed.

The rest of the paper is structured as follows: Section II lists the requirements for an effective COM-stack design. Section III outlines the pitfalls of today's COM-stack designs and why it is not suited to meet the requirements previously stated. Then, our new design concept is presented in Section IV, which is closely related to the middleware implementation to handle object-oriented transmissions. Section V gives an outlook on related work on COM-stack design and Section VI concludes the paper.

II. Requirements for future automotive communication

As previously mentioned, with automated driving the communication fundamentally changes. While today's vehicles are mainly based on small control data (e.g. from an electronic stabilization program), large amounts of data will have to be evaluated at different ECUs in the future. This transforms a car into a data-centric system, containing many time-critical sensor to actuator cause-effect chains whose communication is based on Ethernet [9]. In such an approach, large data objects are the interfaces between different functional units [10]. Nevertheless, prior requirements regarding safety and security still play a role.

In the following, we will gather relevant requirements that are closely related to the safety considerations for Ethernet-based communication in the automotive domain. In accordance with ISO 26262 [1], a major safety objective is the independence from interference. On a technical layer this implies, that the latency of a time critical communication has to be "sufficiently independent" [1], such that interference originating in any lower critical function can be safely bounded. Besides that, data loss during transmission has to be prevented. This concerns all parts of the Ethernet communication, including the middleware, COM-stacks and switches. Together with the challenges of object-oriented transmission and the data-centric and flexible configuration of the car, this results in the following requirements.

Requirement R1: Temporal isolation - predictable timing behavior of high critical traffic. In terms of Ethernet, temporal isolation means, that the timing interference due to lower critical transmissions is bounded. Besides the switches (which enjoyed great research attention [2],[3]), the COM-stack is a processing resource that has to be shared between all traffic streams. To ensure temporal isolation, the COM-stack therefore needs to sufficiently separate packet transmissions of different criticality, both on sending and on receiving. This can either be done priority based or time based (e.g. TDMA), while the latter has shown not to scale well for a high amount of data streams [3]. Consequently it is important to include methods that are able to ensure an upper bound for the interference on high criticality applications.

For packet-oriented transmission, temporal isolation has to consider all packets representing an object. This fundamentally affects worst-case considerations as well as reliability and retransmission patterns, because the classical analytical approaches like compositional performance analysis (CPA) [11] focused on unidirectional single packet transmission times.

Requirement R2: spatial isolation - bounded memory interference. Parallel to the competition for time, critical and non-critical traffic streams compete for memory resources on both, COM-stacks and switches. Especially for an object-oriented transmission, it has to be made sure that the COM-stack buffers as well as output-port buffers of switches do not overflow and loose data especially in the case of a bursty transmission.

Requirement R3 (Resilient Communication): In object-oriented communication, packets might get lost. Like a deadline violation, this might be acceptable for control applications [12]. In contrast to that, applications like sensor fusion require all packets to arrive, preventing image corruption which is crucial to detect obstacles in time. Communication must be

resilient coping with various transient network error scenarios as part of regular operation. This requirement is a consequence of the comparably high bit error rates of practical TSN implementations [13]. Transient errors can be addressed by time-bounded retransmission or redundant transmission. Because TCP incurs too much protection overhead, light weight predictable communication protocols are needed. If fail operational behavior must be guaranteed, a possibly degraded service for permanent network errors has to be provided [13].

On top of these basic requirements, there are new ones from data centric designs and dynamic system behavior. A car which is driving in the city has different requirements to the quality of its sensor data (e.g. frame rate and resolution) or the decision process, depending on car speed and distance to obstacles of different types. These different requirements are typically grouped into system modes.

Requirement R4 (Supporting dynamic data centric design): In a data centric approach, a core platform function is the synchronization of data bases between different units. A popular example is the Data Distribution Service (DDS) [10], and SOME/IP [8] can be used for the same purpose. Synchronization entails communication with the consequence that the communication latency is restricted by the maximum permitted time deviation of data bases. Such a restrictive latency timing approach is currently not part of DDS or SOME/IP, but bounded time deviation of data bases would be very helpful for information fusion or decision making. Moreover, synchronization is on application demand and, as such, subject to the dynamics of system modes.

Requirement R5 (Modularity and Scalability): Modularity is required to flexibly support communication dynamics. There should be an application programmers interface (API) to set up communication channels together with formulating latency requirements for objects and their synchronization. It should be possible to verify adherence to these requirements by exclusively verifying the communication service in order to develop and modify the ECU software independently from the communication services. Scalability requirements include both the number of ECUs and the internal structure of ECUs that is increasingly hierarchical (core-multicore-multichip-...).

III. Pitfalls of today's communication stack designs

In the following section we will take a look at today's COM-stack designs as well as common implementation concepts and show how they fall short on the requirements of Section II when it comes to future automotive communication scenarios. The general flow of Ethernet packet transmission is shown in Figure 1, that is specifically discussed in the following subsection. Note that some of those pitfalls already existed for packet-oriented transmissions but now become urgent problems with the shifts towards object-oriented transmissions (Fig.1:**E**).

A. Periodic execution model

For the automotive domain, AUTOSAR defines the industrial standard for the software architecture [14]. Within AUTOSAR, code generation is used to create runnable entitys (REs) from functional blocks.Those REs are later mapped to container tasks, in which they are executed sequentially. Since automotive software often consists of control applications, most container tasks are activated periodically [15].

The same process applies to the classic AUTOSAR COM-stack, whose REs are typically integrated in a periodic COM-stack task with common periods of up to 5ms (Fig.1:**A**). The stack then first executes all input REs, then passes and receives all data to and from the applications and finally processes all output REs. This has multiple drawbacks. First, the periodic execution enforces a sampling delay, since network packets arrive sporadically and may have to wait to be processed in the next COM-stack period. This problem becomes worse when introducing object-oriented transmissions in a data centric design, where it is likely that some of the packets miss the sampling moment and are therefore processed in the next period. Consequently, the latency is dominated by the sampling delay (multiple ms), obviating optimizations in the network that are typically in the order of 10s to 100s µs.

The second drawback is a high variance in the response time of the COM-stack together with a larger buffer size. In the best case there is no data to process but in the worst case, all REs in the COM-stack task need to process the maximum amount of data that is accumulated during the period. Consequently, buffer sizes need to be larger to hold all packets that might arrive during the COM-stack period. As a result, it is no more affordable to execute the whole COM-stack periodically. Especially with SL LET a technique is available that allows to mask an event based COM-stack for LET based applications.

B. Ensuring data consistency by single processing task

Enforcing consistency for COM-stack internal data can be typically done in two ways. Either the COM-stack is executed in the context of the application software and semaphores are used to protect shared variables [16], or the whole COM-stack is executed in a single task on system level. The lightweight IP (lwIP) stack [17] is one of the most popular stacks for embedded systems and uses the latter approach. Although a single COM-stack tasks allows to run applications with limited permissions which is beneficial in terms of spatial isolation, it is a bottleneck for multiple data streams. The lwIP uses a central FIFO queue that is filled with all sending requests and received packets, making it impossible to distinguish between critical and non-critical data streams (Fig.1:**A**). Especially when we ask for a latency bound of a whole object transmission, missing prioritization results in a pessimistic worst-case assumption. Consequently a novel COM-stack design has to support priority-based queuing to limit this effect.

C. Shared resources

Lightweight COM-stack implementations typically use zero-copy processing to reduce the runtime overhead. A packet

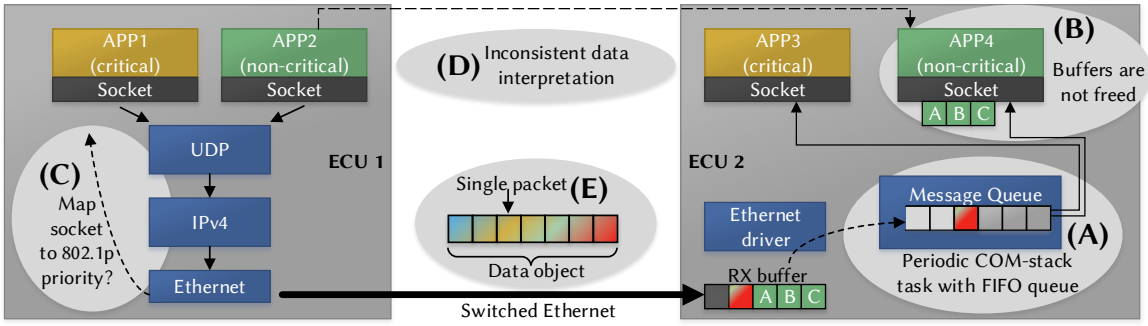

Fig. 1: Common pitfalls in COM-stack designs

is represented by a buffer pointer, which is passed between the different processing stages instead of copying the data itself. In the receive path, a small amount of buffers is allocated in advance and filled on packet reception using direct memory access (DMA). Consequently the spatial isolation is violated as soon as an application does not free received buffers and a buffer underrun occurs (Fig.1:**B**), since this will prevent foreign applications from receiving data. This is even possible if an application opens a socket connection but does not perform read operations, since sockets typically buffer incoming data. Consequently, monitoring is required to ensure that the COM-stack is able to detect such malfunctions.

D. From sockets to priorities

Ethernet has the ability to distinguish between eight different traffic classes for prioritization (IEEE 802.1p [18]). This is an important feature to ensure sufficient temporal isolation between critical and non-critical traffic. On the other hand, applications use sockets to manage their connections, described by IP addresses and UDP/TCP ports. The COM-stack is now in charge of translating socket connections to 802.1p priorities. Lightweight embedded COM-stacks typically do not support such a feature, since there is no direct correlation between the transport and the data link layer (Fig.1:**C**). Acting as a hypervisor in this case, the COM-stack has multiple options to do such a mapping. The first method which is optionally supported by all COM-stacks is to do a packet inspection in the Ethernet driver to identify the correct priority. Besides user-configurable hook functions in lwIP [17], Linux supports this feature by using firewall rules. However, it is not preferable to do this in software due to a high runtime overhead. Another option is to assign one virtual interface per priority, each identified by a unique IP address. This is a non-intuitive correlation between the Ethernet layer and the IP layer, making migration more complex. An example would be a modified sender priority due to a mode change. This would require to inform all receivers that their communication partner has changed its sender address.

A solution to this problem is to insert an intermediate layer between the application and the socket API, providing the COM-stack necessary information to track the packet. Although this is available in the Linux COM-stack, it is implemented with complex lookup tables (net_prio cgroups [19]) since it was not designed from the real-time perspective. For embedded devices, a more efficient lookup is required.

E. Application interface complexity

By accessing the COM-stack directly from the application through a UNIX socket interface, flexibility is very limited and migrating the software is a complex task. At this point, serialization as well as interpretation (Fig.1:**D**) of the transferred data is part of the application and therefore this information has to be propagated offline, e.g. between different vendors. Moreover, the COM-stack has no ability to interpret the data, which leads to problems when large objects are transferred. For large data, the application has to provide a method for fragmentation and reassembling which again has to be consistent on both endpoints. Using a middleware is beneficial in this case, since it removes the burden from the applications and is able to negotiate data types as well as perform compatibility checks. SOME/IP-TP as a relevant automotive middleware [8] is able to perform this type of high-level segmentation and reassembly on the basis of UDP. Hard real-time guarantees or reliability mechanisms such as retransmission of lost packets are out of scope here.

F. Complexity of existing middlewares

The middleware is an additional part that is inserted between the COM-stack and the application. Consequently its timing behavior has direct impact on the E2E timing of cause-effect chains. In addition to SOME/IP-TP, which has too limited functionality, other existing middleware implementations such as the robot operating system (ROS) [20] are insufficiently suited for integration into small embedded systems. They typically use dynamic memory allocation during runtime, leading to a less deterministic response time and include a large amount of quality of service (QoS) parameters that are not necessarily needed. Therefore the library size increases, which is unfavorable in case of restricted resources. In contrast, a middleware should be reduced to a minimum required for interoperability and it should be closely related to the COM-stack, avoiding unnecessary overhead, e.g. due to copy operations.

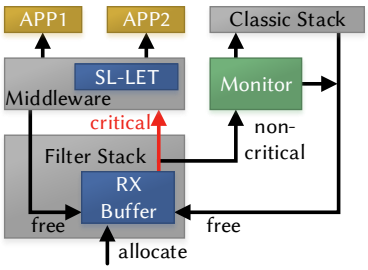

Fig. 2: RX path: Small filter stack combined with middleware

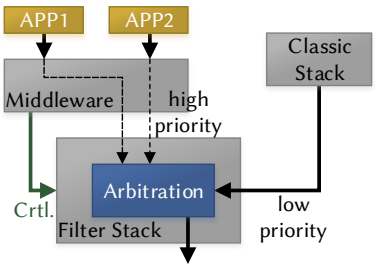

Fig. 3: TX path: Deterministic arbitration and prioritization

IV. Design of a deterministic communication stack

In the following section we outline the design of an efficient and small communication back-end, closely combining a COM-stack with a middleware. It is important that both, the COM-stack and the middleware are considered in combination to exploit the full potential of the application properties. As an example, the middleware has knowledge about the data object that is split in multiple packets and knows how to fragment and reassemble it, making IP fragmentation obsolete. In conjunction with the explicit deadlines of SL LET, it is able to identify necessary retransmission in case of packet loss and to decide when the application needs to be informed in case of a deadline miss, fulfilling Requirement R4 (dynamic data-centric design)

Taking shortcomings of existing COM-stack implementations into account, the design consequently separates between a reception (RX) and a transmission (TX) path. Both paths are activated event based to prohibit sampling delay. Moreover, they are handled separately, either by different tasks or by arbitration of different processing queues.

In contrast to a fully equipped TCP/IP stack, processing a single UDP packet with a reduced lwIP stack can be done much faster. We have done first measurements on a ARM Cortex-R5 (closely related to an automotive R-Car H3 [21]), that show that latencies of around 40µs for the reception and 25µs for the transmission are possible. In contrast to extensions of the Linux stack which are designed to improve the average latency of critical traffic, our approach focuses on short latency guarantees.

SL LET can then be used to integrate such an event based communication back-end in the scope of periodic control applications. By strictly separating communication and computation timing, LET meets the Requirement R5 (modularity and scalability) on a multi-core ECU. SL LET extends that property to the system level in a non-trivial way. It introduces separation between local time deterministic communication and global COM-stack and network communication. The global part takes the role of computation in LET with the important difference of loosely time coupled reading and writing in different time zones [6]. There is no size limitation to the mechanism as the fly-over convention makes the timing independent of the number of involved ECUs and time zones. The same holds for hierarchy, because there is no need to resynchronize at transitions between levels of a hierarchy.

A. RX path

When a new packet arrives, the Ethernet driver is not able to distinguish between critical and non-critical traffic since it does not know about the application context of the data. Consequently, it has to handle each packet as if it is critical. As shown in Figure 2, our design proposes a minimal *filter COM-stack* (e.g. based on lwIP) that is in charge of pre-processing all critical traffic and hand it over to the middleware. But as soon as it detects that the packet is not critical, the packet is passed (unmodified) to a *classic COM-stack* on a lower priority or even to a Linux OS running on a foreign CPU core. This decision can be made on multiple levels. An example would be the reception of a fragmented IP packet which is not supposed to be critical. Instead with the knowledge of the middleware, IP fragmentation can be prohibited for critical traffic by establishing the fragmentation in the middleware. The classic stack then is able to handle the uncritical packets as usual, ensuring backwards compatibility. Due to the reduced complexity, it is possible to fulfill Requirement R1 (temporal isolation). Moreover, the middleware (as a privileged part) ensures that buffers are freed fast enough to prevent a buffer underrun, enforcing spacial isolation (Requirement R2). Whenever packets cannot be freed directly, a monitoring has to ensure that malfunctions are detected (e.g. packets that are passed to the classic stack).

Moreover, the middleware is able to use SL LET for explicit reception deadlines of whole objects instead of single packets. This enables deadline monitoring of object-oriented transmissions and explicit retransmission requests may be issued if there is still enough time left before the deadline.

B. TX path

Figure 3 shows the TX path, where the middleware acts as a hypervisor for all sending applications. It is able to distinguish between applications of different criticalities and can therefore pass priority information in conjunction with the packet buffer to the Ethernet driver, which adapts them to 802.1p priorities on Ethernet. This is an extension to classic socket-based operations, since the additional information can be attached to the socket. For critical traffic, the middleware is able to configure the COM-stack's static address resolution protocol (ARP) tables, making non-deterministic dynamic address resolution unnecessary. However, dynamic ARP can still be part of the classic stack. Due to the detailed

knowledge about whole data objects and their deadlines, future middleware might also able to configure traffic shaping for specific object transmissions, therefore reducing the impact of a packet burst. The underlying COM-stack contains multiple processing queues that are arbitrated according to their priority. As a result, it is also able to handle all Ethernet packets prioritized by the filter stack and providing a best effort output for the classic stack.

This novel design can be easily evaluated in future work, by comparing latency measurements for critical transmissions against a classical TCP/IP stack, as well as by providing a timing model of the filter stack that can be used for verification. The latter only becomes feasible due to the massively reduced complexity of our approach.

V. Related Work

Time deterministic Ethernet communication gained importance in the last years. Standards like TSN have been developed to increase predictability by traffic shaping [2]. Often a time division multiple access (TDMA) behavior is enforced [2] [22], that requires synchronization of time slots for short latency. Unfortunately this concept does not scale well for many competing traffic streams and as soon as slot synchronization can not be guaranteed, normal 802.1p prioritization [18] outperforms those shapers [3]. With SL LET [6], a concept is available that abstracts inter-ECU communication by deterministic LET [23] behavior. Nevertheless, there is still a lack of analysis concepts that provide latency guarantees for the transmission of whole objects, since approaches like CPA focus on single packets [11].

COM-stack design gained less attention in research, although it plays a central role for Ethernet communication. RTnet [22] was developed as a real-time COM-stack for Linux, enforcing predictability by TDMA scheduling on the network. Other solutions that try to put a fast stack in front of the Linux COM-stack like Open Fast-Path [16] or the extended data path (XDP) [19] are often used for fast pre-processing, e.g. for packet inspection or denial-of-service protection. The hard real-time guarantees and a synergy with a middleware are therefore still out of scope. Due to flexible design of the Linux COM-stack, it is a complex task to integrate it into resource constrained embedded systems. In AUTOSAR, the COM-stack has continuously grown by integrating new protocols and supporting legacy systems. Due to the mapping in one task, this leads to a high execution time jitter and introduces unnecessary overhead.

As an upcoming middleware standard, DDS was also introduced in the automotive domain (AUTOSAR adaptive [14]). It is mainly promoted by its ability to scale well and to handle bandwidth-intensive traffic streams such as sensor data [10]. However, DDS provides only a standardized API for accessing QoS parameters that control the communication, but does not give instructions on how to exchange data on a lower level. Therefore, the Real Time Publish Subscribe Protocol (RTPS), which defines the wire protocol, has been standardized [24]. Consequently, for a low-latency real-time DDS, an adequate RTPS implementation must be provided. A well-founded analysis of hard real-time requirements both at the middleware QoS parameter level and in relation to a highly deterministic COM-stack therefore still remains an open issue.

VI. Conclusion

With the trend towards automated vehicles, in-vehicle Ethernet is likely to become the unified communication backbone. High-end requirements move from high-throughput streaming communication with focus on individual packet transport to data-object synchronization requirements in a largely data-centric application environment. We derived the resulting requirements from resilience to end-to-end latencies and argued that SL LET is a suitable programming paradigm to separate ECU design from COM-stack and network design with modularity and scalability thereby keeping control over system timing. We, then outline how a lightweight communication stack with filter functions could serve to reach tight data synchronization.

References

[1] I. ISO, "26262: Road vehicles-functional safety," *International Standard ISO/FDIS*, vol. 26262, 2018.

[2] "802.1Qbv - Enhancements for Scheduled Traffic," http://www.ieee802.org/1/pages/802.1bv.html, accessed: 2019-05-21.

[3] D. Thiele, R. Ernst, and J. Diemer, "Formal worst-case timing analysis of ethernet tsn's time-aware and peristaltic shapers," in *IEEE VNC*, 2015.

[4] "Open Networking Foundation," https://www.opennetworking.org/sdn-definition/, accessed: 2019-05-21.

[5] A. Angermann, M. Beuschel, M. Rau, and U. Wohlfarth, *MATLAB-Simulink-Stateflow: Grundlagen, Toolboxen, Beispiele*, 2014.

[6] R. Ernst, L. Köhler, and K.-B. Gemlau, "System Level LET: Mastering cause-effect chains in distributed systems," in *IECON*, Oct. 2018.

[7] T. A. Henzinger and C. M. Kirsch, "The embedded machine: Predictable, portable real-time code," *ACM Trans. Program. Lang. Syst.*, 2007. [Online]. Available: http://doi.acm.org/10.1145/1286821.1286824

[8] "Scalable service-Oriented MiddlewarE over IP (SOME/IP)," http://some-ip.com/, accessed: 2019-05-22.

[9] R. Ernst, "Automated driving: The cyber-physical perspective," *Computer*, vol. 51, no. 9, pp. 76–79, 2018.

[10] "Whitepaper: The Secret Sauce of Autonomous Cars ," https://www.rti.com/secret-sauce-autonomous-cars-wp, accessed: 2019-05-21.

[11] R. Henia, A. Hamann, M. Jersak, R. Racu, K. Richter, and R. Ernst, "System level performance analysis - the symta/s approach," *IEE Proceedings - Computers and Digital Techniques*, March 2005.

[12] P. Pazzaglia, C. Mandrioli, M. Maggio, and A. Cervin, "DMAC: Deadline-Miss-Aware Control," in *ECRTS*, 2019.

[13] M. Moestl, D. Thiele, and R. Ernst, "Towards fail-operational Ethernet based in-vehicle networks," ser. DAC '16. ACM, 2016. [Online]. Available: http://doi.acm.org/10.1145/2897937.2905021

[14] "AUTOSAR standards," https://www.autosar.org/standards/, accessed: 2019-04-15.

[15] A. Hamann, D. Dasari, S. Kramer, M. Pressler, F. Wurst, and D. Ziegenbein, "Waters industrial challenge 2017," in *WATERS*, 2017.

[16] "Open Fast-Path," https://openfastpath.org/, accessed: 2019-05-22.

[17] A. Dunkels, "lwIP - a lightweight TCP/IP stack," *http://www. sics. se/ adam/lwip/index.html*, 2002.

[18] N. Ek, "Ieee 802.1 p," *Q-QoS on the MAC Level*, 01 1999.

[19] C. S. V. Gutiérrez, L. U. S. Juan, I. Z. Ugarte, and V. M. Vilches, "Real-time linux communications: an evaluation of the linux communication stack for real-time robotic applications," *arXiv preprint arXiv:1808.10821*, 2018.

[20] "Robot Operating System," https://www.ros.org/, accessed: 2019-05-22.

[21] "R-Car H3 System-on-Chip (SoC)," https://www.renesas.com/us/en/solutions/automotive/soc/r-car-h3.html, accessed: 2019-05-22.

[22] "RTnet, Hard Real-Time Networking for Real-Time Linux," http://www.rtnet.org/index.html, accessed: 2019-05-22.

[23] T. A. Henzinger, B. Horowitz, and C. M. Kirsch, "Giotto: a time-triggered language for embedded programming," *Proceedings of the IEEE*, vol. 91, no. 1, pp. 84–99, 2003.

[24] "OMG - Data Distribution Service," https://www.omg.org/spec/category/data-distribution-service/, accessed: 2019-05-22.