Reliable Distributed Systems

Philipp Mundhenk, Arne Hamann, Andreas Heyl, Dirk Ziegenbein

Robert Bosch GmbH

Renningen, Germany

{philipp.mundhenk|arne.hamann|andreas.heyl|dirk.ziegenbein}@de.bosch.com

***Abstract*—The domains of Cyber-Physical Systems (CPSs) and Information Technology (IT) are converging. Driven by the need for increased compute performance, as well as the need for increased connectivity and runtime flexibility, IT hardware, such as microprocessors and Graphics Processing Units (GPUs), as well as software abstraction layers are introduced to CPS. These systems and components are being enhanced for the execution of hard real-time applications. This enables the convergence of embedded and IT: Embedded workloads can be executed reliably on top of IT infrastructure. This is the dawn of Reliable Distributed Systems (RDSs), a technology that combines the performance and cost of IT systems with the reliability of CPSs. The Fabric is a global RDS runtime environment, weaving the interconnections between devices and enabling abstractions for compute, communication, storage, sensing & actuation. This paper outlines the vision of RDS, introduces the aspects required for implementing RDSs and the Fabric, relates existing technologies, and outlines open research challenges.**

***Index Terms*—cyber-physical systems, reliable distributed systems, real-time systems, cybersafety**

I. Introduction

Across all system domains there is a strong trend towards cloud connectivity towards a ubiquitous availability of compute and data. This trend has certainly manifested itself in the business and infotainment domains, e.g., the streaming of music, movies and now also games, thereby shaping also the end consumer devices for e.g., listening to music from CD players over mp3 players to smartphones or tablets. But this trend also affects CPS, e.g. the cloud-based navigation in mobility or post-processing of manufacturing data.

Consequently, the product portfolios in various industrial domains currently evolve from classical non-connected CPS applications (e.g. engine control) to cloud enhanced CPSs that use networking and cloud computing for non-critical functional enhancements [1], e.g. sending monitoring data to the cloud for predictive maintenance. These enhancements have little impact on the original devices in terms of their functionality and scope.

Advancing cloud connectivity one step and enabling cloud or edge cloud platforms to also support the execution of real-time and safety-critical applications with an assured degree of reliability (see Fig. 1), offers a plethora of new possibilities. Such RDSs enable functions which have been tied to the device to become nomadic, i.e. they can be migrated from the device to the (edge) cloud. This will lead to new sweet spots for deploying functionality and will impact CPS devices similarly as described for the infotainment domain above. A vehicle will e.g., require a reduced set of sensors and less compute in order to be capable of urban automated driving because it can rely on capabilities of other vehicles and of the stationary infrastructure (both sensors and compute). Additionally, RDS enable completely new functionalities as well as new pay-per-use business models such as hourly automated driving passes or automated valet parking in urban spaces.

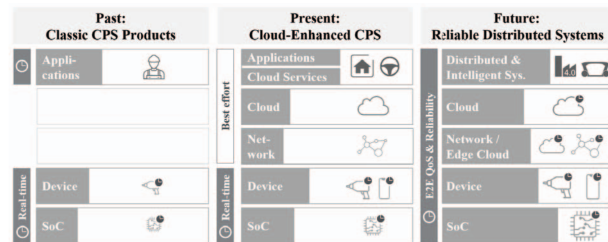

Fig. 1. Evolution of Cyber-Physical Systems: From separate systems, over cloud-connected systems to Reliable Distributed Systems (RDS). The latter are inherently connected and distribute all their functionality in compute, sensing, and actuation.

On the other hand, the complexity and dynamics of the resulting distributed system mandates an autonomic approach to realizing RDS by specifying and publishing capabilities and needs of individual components while having the system largely configuring, supervising and adapting itself autonomously.

A. Applications & Benefits

While many approaches limit themselves to one domain (e.g. [2]), the applications of RDS are limitless. Every embedded system and CPS with connectivity to the internet can benefit from RDS. RDS makes compute power universally available, automatically bookable at runtime. Furthermore, sensors and actuators are flexibly available and can be used by other systems, as required. The following are example applications and benefits from different domains:

- *Infrastructure-Assisted Automated Driving*: With RDS, vehicles no longer require two high-performance autonomous driving compute units. A single, or maybe no unit, is sufficient, as compute power can be used on demand. The Fabric e.g., might contain edge servers, offering low latency compute on city level. This can be used for urban autonomous driving computation. Additionally, sensors can be added and booked by the vehicle on demand, allowing a safety never seen before. By utilizing a security camera attached to a building, e.g., a vehicle is able to perceive objects around a corner and out of sight of the on-board sensors.
- *Smart Factories*: Today, computing units (e.g. Programmable Logic Controllers (PLCs) or industrial PCs) are locally installed at many production machines. These computing units are based on proprietary software stacks and are often in use for decades. Accordingly, they are a hindrance when it comes to a modern IT infrastructure that

can keep pace with the innovations of the smart factory, especially with regard to flexible and convertible production lines. RDS fundamentally changes this situation. Similar to the concept of a thin-client architecture, machines on the manufacturing floor can provide a simple I/O interface to the Fabric that manages logic and high-level control in real-time. This operational shift towards the Fabric consolidates and simplifies software development, deployment, and management resulting in a significant price advantage.
- *Developer Efficiency*: By strict division of concerns and an appropriate system model, application developers can focus on the business logic of an application and do not need to worry about the infrastructure. This effectively increases their development efficiency. Furthermore, by unifying the already similar approaches across domains, developers can be exchanged between domains, alleviating the concerns of a tight labour market.
- *Resource Efficiency*: By utilizing the available compute capabilities, sensors, and actuators, the overall number of devices can be reduced. Rather than every device bringing their own components and many devices sitting idle for extended periods of time, the contained components can be shared among devices. This increases the utilization, thus lowering the overall amount of components required.
- *Privacy Benefits*: At first glance, universal access to sensors and actuators might seem counter-productive to privacy. But using systems in a more distributed manner and offering shared compute performance at a local level, the need to communicate large amounts of data to the cloud is alleviated. The misuse of data is becoming increasingly difficult, if it is widely distributed, thus increasing privacy as an automatic side effect of RDS. As in existing systems, also in RDSs, it is of crucial importance to support user-level access control mechanisms to data and sensors and actuators.

B. Aspects of RDS

RDS consists of multiple aspects. At design time, a system model represents the system, describes its components, and captures their requirements and capabilities (see Section II). This component metadata is essential not only for planning and designing the system but also to be able to control the system at runtime. To pass this metadata to the runtime, e.g., manifests can be used.

At runtime, multiple abstractions are required. Compute capabilities need to be abstracted to allow the sharing of the same (see Section III-A). This is essential to enable the dynamics of nomadic functions, decoupling applications from compute components of particular devices. Similarly, storage needs to be abstracted to allow nomadic functions to access e.g., internal state at their execution location (see Section III-B). The abstraction of communication is required to avoid applications having to control their underlying infrastructure to e.g., request the required data (see Section III-C). Additionally, when interacting with the physical world, as RDS does, the sensors and actuators as interfaces to the physical world need to be abstracted and made available to applications in a generic manner (see Section III-D). Last, but not least, in order to be able to use RDS in safety-critical CPS, it must also offer the possibility to ensure that the underlying distributed execution is functionally safe (see Section III-E).

There are a number of cross-cutting concerns, that are required in each of these aspects. In addition to safety, this also holds for security and privacy. These need to be inherent to each abstraction, thus also satisfying the real-time requirements posed to every abstraction. Existing security approaches might not always be suitable for real-time applications (see e.g., [3]).

II. System Model

An essential part of realizing RDS and the Fabric is the possibility to describe the components of the system as well as their capabilities and requirements at design time. However, the system itself is not defined in its entirety at design time, but rather created from well-defined components at runtime. Components can be pieces of hardware or software. Hardware components are e.g., basic hardware elements, such as CPU, GPU, but also more complex devices, such as System on Chip (SoC) or devices containing hardware components. Software components can be infrastructural software components, such as Operating Systems (OSs), middleware, communication systems, etc., as well as application software components. Components are hierarchical and may be grouped into a component. Component requirements are posed to lower layer components, whereas capabilities are offered to higher layer components. Both, requirements and capabilities can be functional or non-functional requirements and thus include e.g., processing performance & type, memory requirements, but also redundancy for safety, secure storage & execution, etc.

In addition to lower layer requirements, components may offer or declare dependencies on data, which, in contrast to a vertical layer requirement, is a horizontal requirement. Such data dependencies are typically fulfilled on the same or an immediately adjacent layer (e.g., in case of services). Note that dependencies are always on data, never on other components, allowing for lose coupling in the system. Data definitions can be based on existing standards, such as VSS [4] or custom defined.

The system model does not necessarily contain a deployment, i.e., which higher layer component is deployed to which lower component in the system. While this might be defined for certain less flexible components, such as hierarchical hardware components, e.g., the hardware of a 5G base station, it is not mandatory. This enables the required degrees of freedom at runtime to e.g., optimize, and react to failures.

The requirements and capabilities of individual components are matched by the runtime environment. For reliability, this is performed in a decentralized and potentially, in critical environments, a redundant manner. Considering a large RDS, the solution space will be very large, requiring significant computational performance to reach optimal results. It is noteworthy, that the runtime environment on a single component (e.g., a microcontroller-based system with limited performance), does not necessarily need to perform the required computations itself. Instead, the individual component might operate on pre-computed deployment plans, which contain likely failures and are pre-computed in a connected backend (e.g., cloud). Once the current state of the system leaves a defined safety-envelope in the pre-computed deployment plan, the computation of a new deployment plan in the backend is triggered. This is

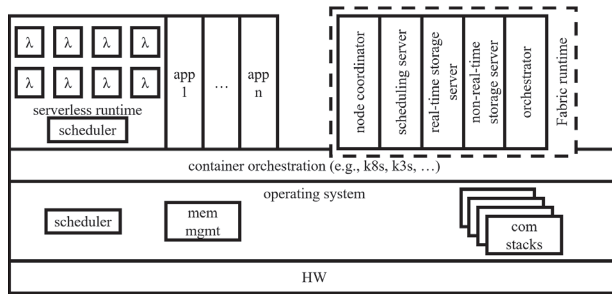

Fig. 2. Simplified architecture of a Fabric node. Note, all required services being implemented as containers, just like applications. Also note, serverless functions running in a dedicated container with dedicated scheduler (hierarchical scheduling).

possible, as we are assuming an inherently connected system. The computation of the deployment plan itself is of course also a nomadic workload as all others in the systems and can be scheduled on any suitable component.

III. Runtime Environment

A. Compute

Migrating computational workload between compute components is an essential aspect of RDS. To allow for devices to offer their computational capabilities in a Fabric, an abstraction is required. Multiple dimensions of compute abstractions can be imagined. Initial implementations of RDS will include virtualization techniques for homogeneous compute modules (see Fig. 2). Heterogeneous compute abstractions, bridging, e.g., Central Processing Unit (CPU) and GPU, are beyond the initial scope of the Fabric and may be supported later.

1) Virtualization Techniques: Virtualization is key to portability of computational workloads. A number of virtualization techniques exist [5], the suitability of candidates relevant to RDS is discussed in the following.

- *Emulation* translates (with optimizations) the Instruction Set Architecture (ISA) of a target application to the ISA of the underlying hardware. This enables the execution of binaries compiled for a different ISA (e.g., ARMv8 on x86). Emulation is out of scope of RDS at runtime. It is assumed the correct binaries are available. Emulation may be used for development purposes.
- *Hypervisors* allow the virtualization of virtual machines including operating systems and applications with suitable ISA on hardware (Type 1) or a host OS (Type 2). While hypervisors might be used in the underlying layers of RDS, e.g., to manage shared resources, on which the Fabric is executed, they are not a key aspect of RDS.
- *Containers* are a lightweight virtualization technique for applications (including dependencies), building on the host OS kernel. Containers will be an essential technology in RDS compute abstractions. As containerized applications are executed as OS processes on the host (with limited scope and permissions), the extensively researched domain of real-time scheduling can be applied.
- *Serverless (Lambda)* functions are even more lightweight than containers, relying on the runtime environment to execute a single process and supply dependencies. Just like containers, serverless functions are a key technology in RDS. The suitability of serverless functions for hard real-time applications is an open field of research.

The Fabric will thus contain a mixture of containers and serverless functions. Due to the expected heterogeneous nature of the Fabric, different compute hardware types (CPU, GPU), as well as different ISAs will be prevalent. The runtime needs to ensure that the correct binary format is executed, which need to be provided by the continuous integration pipelines. This challenge is largely solved for connected systems in the domain of cloud computing.

2) Scheduling & Distribution: While execution management in cloud computing is mostly optimized for high throughput, dealing with real-time systems also mandates to achieve bounded latencies. As processes in containers are scheduled as processes on the host OS, the domain of real-time scheduling applies here. However, the Fabric will typically consist of Linux-based OSs, which, due to their complexity (interrupts, etc.), make scheduling tasks significantly more hard to compute. These challenges are addressed in a number of other works [6], [7]. Recently, Real-Time Operating Systems (RTOSs) started advertising support for containerization [8]. This may prove itself as a valid opportunity for virtualization of hard real-time critical applications.

In serverless computing, typically, an additional scheduling and orchestration plane is introduced. The amount of literature on serverless function scheduling is limited [9]. However, this is essentially a hierarchical scheduling problem [10].

An additional challenge is being posed by the distributed nature of the system, as well as the safety concepts. This may require redundancy and software lockstep for relevant applications, not trivially implemented efficiently in distributed systems. While redundancy is common in cloud environments, the time constants and resource constraints in embedded systems differ significantly. An additional clone of the application can often easily be afforded in cloud environments to offer higher availability. On embedded systems, on the other hand, the amount of resources is highly limited and these need to be used efficiently. At the same time, the tools and methods used in cloud computing are fairly slow in terms of setup and synchronization times, when compared to the requirements of hard real-time systems. An optimal solution needs to balance the longer lead times and real-time requirements with the resource limitations and redundant active compute workloads.

B. Storage

Distributing functionality across devices poses the challenge of having relevant data, such as internal state, available on all executing nodes in synchronization. This is amplified when working with real-time critical applications in fail-operational concepts, as hot spares need to be synchronized at all times. The general problem of synchronizing distributed storage is well studied and available as products in the domain of cloud computing [11], [12]. While the general concepts may also apply to the Fabric, the field of (hard) real-time storage synchronization is still in its infancy. Some work exists in the domain of fail-operational systems [13]. These are, however, usually based on hot spare approaches, having identical, but passive compute workloads recreating the internal state from identical inputs. A passive synchronization of storage including real-time

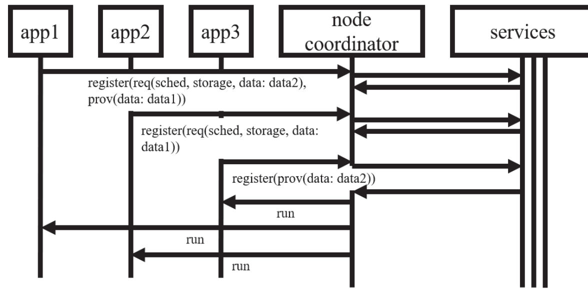

Fig. 3. Simplified application startup sequence. Apps are registering with requirements (*req*) and provided capabilities (*prov*) at node coordinator and, transitively at other services, such as scheduler, storage server, etc. When requirements are fulfilled in the system, apps are started. Note that orchestration may also happen across nodes.

critical internal state, as known from cloud environments, is not existent.

C. Communication

In distributed systems, communication is essential to the functionality of the system. All devices in the space of RDS are inherently connected. Connectivity, however, can be viewed on multiple layers: From individual voltage levels on a physical layer, to the universal distribution of data. In RDS, applications operating on the Fabric need an abstracted view of communication. Applications have requirements on data and its functional and non-functional properties, such as resolution (spatial/time), unit, quality/accuracy, age, etc. The task of the Fabric is to deliver the required data with the required attributes (see Fig. 3). This of course assumes that the data is existing in the Fabric and requires exception handling (e.g., start-up prevention), if mandatory requirements such as data are not fulfilled.

Identification of data in distributed systems is non-trivial. In existing systems, typically dedicated identifiers are agreed upon, which are centrally managed. In case of MQTT or DDS these identifiers are called topics, in case of SOME/IP it uses numerical service identifiers. In a fully distributed system, a central data identification format is impractical. Thus, a distributed approach to managing data identifiers needs to be developed. A potential approach could be the organization of the Fabric in areas based on geolocation with dedicated, redundant coordination points, effectively distributing central data identification and reducing the direct interconnectivity. This approach is referred to as spatial computing [14]. Alternative approaches are the standardization of communicated information, basically equivalent to an Application Programming Interface (API) with semantics. One such approach in the domain of vehicles is Genivi Vehicle Signal Specification (VSS) [4].

D. Smart Sensors & Actuators

CPS are characterized by their interaction with the real world. Input from that environment is implemented through sensors, manipulation of that environment is implemented through actuators. Often, sensors and actuators are directly connected to computational units in a distributed system. Standalone sensors and actuators are often called smart sensors & actuators, containing the possibility to perform e.g., pre/post-processing of input/output data. Additionally, these devices are able to communicate with the system without the need for a dedicated compute unit. Rather than sending/receiving individual sensing/acting values and assuming the receiver is aware of the context of this data (e.g., unit, resolution, age), such non-functional data is included in the transmission from the sensor/to the actuator. This benefits the usage of the data in a global RDS onthology and delivering the correct data to sensors. In combination with nomadic compute workloads, this reduces the need for direct local interconnects of sensors and actuators to compute units. Furthermore, the introduction of smart sensors and actuators makes their sensed data and offered actuation services available to a broader range of devices.

E. CyberSafety

For the use for safety-critical applications RDS has to be able to provide dependability for at least an application-specific subset of functions. Dependability requirements affect the integrity of data and execution, as well as the reliability, availability, and also maintainability of components and resources in a frequently changing technical environment.

Dependability is not trivial in distributed systems. Due to non-deterministic and aperiodic scheduling, dynamic resource allocation and dynamically changing interfaces, they are on a technical level inherently complex systems.

The fundamental distributed approach implies from the outset that safety cannot be effectively managed in a central, monolithic manner on platform level. Instead, RDS will follow a modular and multi-layered safety approach. This approach comprises modular safety building blocks on lower platform levels implementing basic technical safety measures, e.g., diagnosis, and providing safety data and dependability meta-data, like data properties, component and platform capabilities, through standardized interfaces (API) to higher layers of the system. Based on this input, higher layers can implement safety mechanisms, e.g., orchestrators using multiple data sources and redundancy concepts. As known from other well-proven safety design solutions [15] the technical measures are complemented by system-level functional safety mechanisms. These additionally perform, e.g., application-specific physical plausibility checks, ideally being unaffected by the events happening at the lower technical levels.

This modular and layered approach allows evolution by facilitating the possibility to upgrade and interchange components while also taking into consideration the physical part of the CPS system level from the beginning.

The particular view of safety on the aspects discussed in this paper is as follows:

- *Compute:* Since safety is not one of the required properties of the cloud technologies used by RDS one cannot expect these technical components to come with certified safety properties like a safety integrity level [16] ensured at design time which would be a typical prerequisite for safety design in other domains, in particular in the automotive industry. Instead, RDS will aim to use known and established safety mechanisms for the use of Commercial-Off-The-Shelf (COTS) components in safety-critical applications, e.g., software lockstep, majority voter, or protective wrapper [17], to assess and assure safety on a

higher technical layer. This enables independence from the concrete implementation on compute level.

- *Storage:* The same predicament also applies to storage. However, the distribution of data and ensuring the integrity of the used storage is a well addressed problem in cloud computing [18]. Mechanisms that are already available can be used for safety argumentation. These should ideally be supported by reliability data and ideally by online assessment of the resulting data integrity. This can be achieved by making the underlying mechanisms and current health states visible to the higher technical layers.
- *Communication:* Data that has been 'safety-assured' at one place within the Fabric has to be distributed to the components that depend on it while ensuring no undetected data corruption can occur. For this purpose RDS will make use of known and established end-to-end safety mechanisms, e.g., checksum, sequence counter, timestamps, and timeout monitoring, that allow to black-channel large parts of a non-reliable communication path [19]. This can again be supported by higher-level redundancy concepts utilizing redundant data and communication channels.
- *Smart Sensors and Actuators:* Within the Fabric sensors can be dynamically included to provide necessary, possibly geolocal, information. This can happen on a higher system level in a plug'n'play fashion based on the data context given in the global RDS ontology and the dependability capabilities and current status communicated via the sensor's interface. For safety-critical actuators to be used within the Fabric, they must have the property of intrinsic safety, thus being able to safely and independently reach a safe state (considered on system level) if the connection to the rest of the Fabric is corrupted or lost.

Having to accept the dynamic complexity of the underlying system and the limited reliability of its individual components, one of the safety paradigms of the Fabric will be to achieve a high degree of fault tolerance and robustness across all technical layers. This is achieved in particular by means of data and design diversity to provide the required dependability on system level.

One key technical concept enabling this will be the establishment of a continuous self-awareness of components and systems within the Fabric. The Fabric needs to be aware of the current technical and functional (e.g., for Safety of the Intended Functionality (SOTIF) [20]) dependability capabilities. Furthermore, it requires the ability to both act on it, e.g., through mitigation or fallback within the defined fault tolerant time interval, and provide this as information to other entities in the network, e.g., via dependability metadata at the interfaces. This allows the construction and online evaluation of dynamic safety assurance cases.

IV. Conclusion

The domains of Cyber-Physical Systems (CPSs) and Information Technology (IT) are converging. Reliable Distributed

Another important aspect will be the ability to dynamically evaluate the dependencies of the supposedly diverse and non-dependent components used in redundancy safety mechanisms. It is the distribution aspect of RDS that supports a higher degree of non-dependence, or freedom from interference [16], of the elements involved, which might not only be technically, but also physically isolated.

Systems (RDSs) are the technological basis for executing embedded, real-time-, and safety-critical compute loads on COTS hardware and software layers. The Fabric is a single, global RDS runtime environment, connecting compute nodes, sensors and actuators in a dependable way on a never before seen scale, significantly extending what is currently understood as a CPS. This paper outlines the required aspects and technological building blocks required to build the Fabric. Many of these building blocks are already available today, some require the integration of technology from other domains into RDSs, and others are open research challenges. Key building blocks and the Fabric are currently under development at Robert Bosch GmbH Corporate Research.

References

[1] A. Hamann, S. Saidi, D. Ginthoer, C. Wietfeld, and D. Ziegenbein. Building end-to-end iot applications with qos guarantees. In *57th ACM/IEEE Design Automation Conference (DAC)*, 2020.

[2] P. Mundhenk, E. Parodi, and R. Schabenberger. Fusion: A Safe and Secure Software Platform for Autonomous Driving. In *2nd International Workshop on Autonomous Systems Design (ASD 2020)*, volume 79 of *OpenAccess Series in Informatics (OASIcs)*, pages 2:1–2:6, Dagstuhl, Germany, 2020. Schloss Dagstuhl–Leibniz-Zentrum für Informatik.

[3] P. Mundhenk. *Security for Automotive Electrical/Electronic (E/E) Architectures*. Dissertation, Technische Universität München, 2017.

[4] Connected Vehicle Systems Alliance. Vehicle Signal Specification. https://github.com/COVESA/vehicle_signal_specification. Accessed: 2021-12-10.

[5] N. Manohar. A survey of virtualization techniques in cloud computing. In *Proceedings of International Conference on VLSI, Communication, Advanced Devices, Signals & Systems and Networking (VCASAN-2013)*, pages 461–470, India, 2013. Springer India.

[6] L. Abeni, G. Lipari, and J. Lelli. Constant bandwidth server revisited. *SIGBED Rev.*, 11(4):19–24, January 2015.

[7] J. K. Strosnider, J. P. Lehoczky, and Lui Sha. The deferrable server algorithm for enhanced aperiodic responsiveness in hard real-time environments. *IEEE Transactions on Computers*, 44(1):73–91, 1995.

[8] Wind River. VxWorks The Leading RTOS for the Intelligent Edge. https://www.windriver.com/products/vxworks. Accessed: 2021-12-10.

[9] A. Suresh and A. Gandhi. Fnsched: An efficient scheduler for serverless functions. In *Proceedings of the 5th International Workshop on Serverless Computing*, WOSC '19, page 19–24, New York, NY, USA, 2019. Association for Computing Machinery.

[10] I. Shin, A. Easwaran, and I. Lee. Hierarchical scheduling framework for virtual clustering of multiprocessors. In *2008 Euromicro Conference on Real-Time Systems*, pages 181–190, 2008.

[11] Rancher. Persistent Storage with Longhorn. https://rancher.com/products/longhorn. Accessed: 2021-12-10.

[12] Red Hat, Inc. Ceph.io. https://ceph.io/en/. Accessed: 2021-12-10.

[13] P. Weiss, A. Weichslgartner, F. Reimann, and S. Steinhorst. Fail-operational automotive software design using agent-based graceful degradation. In *2020 Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 1169–1174, 2020.

[14] C. Lathan and G. Ling. Spatial Computing Could Be the Next Big Thing. https://www.scientificamerican.com/article/spatial-computing-could-be-the-next-big-thing/. Scientific American, 2020-11-10.

[15] EGAS Workgroup. Standardized e-gas monitoring concept for gasoline and diesel engine control units version 6.0. Technical report, 2015.

[16] International Organization for Standardization (ISO). Road vehicles – Functional safety. Norm ISO 26262, 2018.

[17] F. Ye. *Justifying the use of COTS components within safety critical applications.* PhD thesis, University of York, UK, 2005.

[18] K. V. Vishwanath and N. Nagappan. Characterizing cloud computing hardware reliability. In *Proceedings of the 1st ACM Symposium on Cloud Computing*, SoCC '10, page 193–204, New York, NY, USA, 2010. Association for Computing Machinery.

[19] AUTOSAR. *E2E Protocol Specification AUTOSAR*, Nov 2020. Release R20-11.

[20] International Organization for Standardization (ISO). Road vehicles – Safety of the intended functionality. Norm ISO/PAS 21448, 2019.