Software System of Autonomous Vehicles: Architecture, Network and OS

University of Pennsylvania, School of Engineering and Applied Science

Student: Qiaochu Guo, qiaochug@seas.upenn.edu
Project Advisor: Prof. Boon Thau Loo, boonloo@cis.upenn.edu
Course Coordinator: Prof. Max Mintz, mintz@cis.upenn.edu

Date: April 29, 2020

Introduction

Autonomous-vehicle (AV) is "a vehicle that has the capability to drive without the active physical control or monitoring by a human operator" [96]. There are 5 levels of autonomy. Lower level (level 0-3) of autonomy systems consist of multiple individual ADAS's (Advanced Driver Assistance Systems [98]) that help the driver with specific tasks, such as parking, lane-tracking and braking. Higher level (level 4-5) autonomy systems, however, aim to completely replace the human driver, thus requiring a higher level of machine intelligence, which is often provided by neural network algorithms, and a higher level of information integration and functional coordination capability [97]. The Boston Consulting Group estimates that by 2030, fully autonomous vehicles will make up 10% of vehicle sales around the world, and 44% of U.S. drivers surveyed in the study plan to purchase a fully autonomous vehicle within the next decade [99].

As the major automotive companies and technology companies (GM, Ford, Daimler, Google, Amazon, Uber etc.) race to present the world with mass market autonomous vehicles [1], the technologies around autonomous driving have been developing quickly in research institutes as well as in industry labs. Specialized disciplines in perception, planning and control systems are being studied in depth and the techniques are advancing fast. This thesis, however, focuses not on each specialized algorithms, but on the system design aspect of autonomous vehicle as a whole.

The anatomy of an autonomous-vehicle software system is quite different from that of a PC. It usually consists of dozens of ECUs (Electronic Computing Units) that are like mini-computers, each in charge of a separate functionality, such as vision, radar, steering, braking and infotainment [34]. They make decisions either cooperatively through a central "brain" or individually in a distributed manner, depending on the level of autonomy and the chosen design pattern [34].

At the current state, there are usually 60-100 ECUs on a single vehicle, and they often operate on as many as 6-8 different operating systems [95]. This makes the code base bloated (>100m lines) and incompatible for updates [34]. Therefore, an overhaul of architecture design to increase system efficiency is due. In the ideal scenario, it is estimated that there could be as few as 6-10 consolidated ECUs on each vehicle, operating on the same software platform in the future [95]. Accordingly, new software solutions need to be developed to enable the transformation, such as more versatile OSes and service-oriented communication [22].

In this thesis, we will first examine the architecture design from two perspectives: "software architecture" [6, 22, 24] and "functional architecture" [1, 45, 48]. Respectively, we will examine the development stack design (OS, communication network, middleware, and applications), as well as how the top-level applications divide up the required functionalities of an AV system and coordinate to work together. Throughout the discussion, architecture designs will be judged on their software "quality attributes" [7] such as interoperability and modifiability, and evaluated for their implications on the competitive landscape of the industry. After the survey of architecture designs, we will dive into the two foundational infrastructure for the AV software system – communication network (vehicle bus system [11]) and operating system, to gain a more zoomed-in picture of challenges faced by AV software developers on the system design aspect. Product offerings of these two infrastructure will be analyzed on their design, impact on the AV industry, and future trajectories.

Architecture Design

Overview

The Importance of Architecture

"Software architecture was introduced as a means to manage complexity in software systems" [45] and autonomous vehicles are very complex systems. Industries producing similarly complex products with stringent performance requirements usually have standardized software architecture, such as Future Airborne Capabilities Environment (FACE) in the aerospace industry, which helps increase modularity and portability, so that different teams could parallelly develop components that easily integrate together [49].

Besides, though safety standards specifically designed for AVs are not well-established [51], there does exist functional safety standards developed for automotive in general, such as ISO26262, the compliance to which is mandatory for all road vehicles. Compliance to ISO26262 adds significant burden to software developers [50] and such compliance can usually be much more easily proven if the industry shares a common functional and software architecture, and can therefore reuse tested codes [49].

Therefore, an effort to standardize AV architecture will not only speed up development by allowing more parallelization and specialization in between teams, but also helps the industry compound safety knowledge in an area where safety is yet undefined.

State of Literature

The state of literature on AV architecture mirrors that of AV safety standards. Though the specific algorithms used for key autonomous driving problems, such as object detection, localization and trajectory planning, have been under intensive research for decades, the literature on the architecture design aspects of AVs have been lacking. Concern over this lack of attention has been voiced by multiple researchers that are beginning to touch upon this field over the past few years [6, 43, 44, 45].

As the survey for this thesis progresses, it became clear that among the limited existing literature, there is also a lack of a common research methodology for architecture design, with varying degrees of focus on quantitative experimentation versus qualitative reflection. Granted, two popular presentation approaches have been identified by Serban et al. (2018) : (1) "proofs-of-concept from experiments … or competition" (2) "high level overview of system components" [45]. For example, among the papers discussed in the following section, [6, 22, 23, 44, 46, 47] all presented details on competition implementations, and [1, 20, 45] presented high level overviews of components. However, even among papers that took the same listed approach, there are significant differences in terms of topical focus and use of terminologies.

Discussion Organization

In an effort to establish structure, this thesis organizes the discussion on architecture around six topics identified by the author based on their perceived importance in the surveyed papers -- instead of around a commonly agreed component list, which does not seem to exist yet in the early-stage

literature. (The component identification differences will be discussed in the "functional distribution" section.)

It is also worth clarifying that two groups of issues fall under two orthogonal perspectives of AV architecture design -- "functional architecture" [1, 45] and "software architecture" [22, 24] (or "software infrastructure" as in [6]). Though these two terms are sometimes used interchangeably in the literature, we adopt the following definitions for clarity in this thesis.

"Functional architecture": Both Behere et al. (2016) and Serban et al. (2018) define "functional architecture" in reference to ISO26262 functional safety standard's definition of "functional concept" [1, 45], which is "specification of the intended functions and their interactions necessary to achieve the desired behavior" [45, 48]. Therefore, "functional architecture" refers to "the logical decomposition of the system into components and sub-components, as well as the data-flows between them" [1]. The graph below gives an example of a functional architecture design for AV.

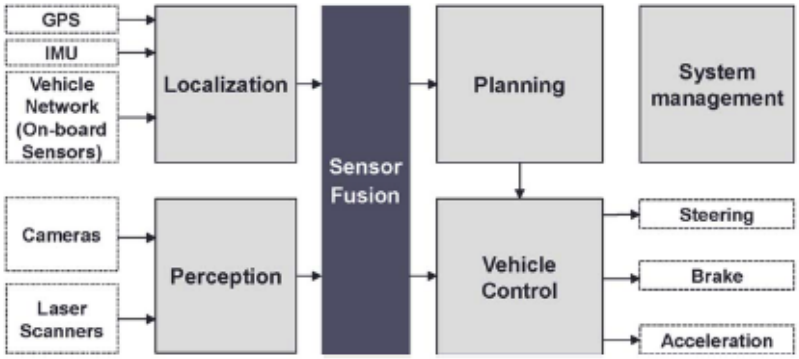

Fig 1. Functional Architecture [23]

"Software architecture": Software architecture refers to the layout of infrastructure components that define their responsibilities and relationships. The infrastructure components usually include I/O devices, operating system, middleware and application modules. According to Mcnaughton et al. (2008), the functional architecture that defines relationships between the planning, perception and world modelling modules, as introduced above, should be considered as built "atop a software infrastructure" [6]. The graph below gives an example of a software architecture, which describes the same AV as shown in the graph above depicting its functional architecture.

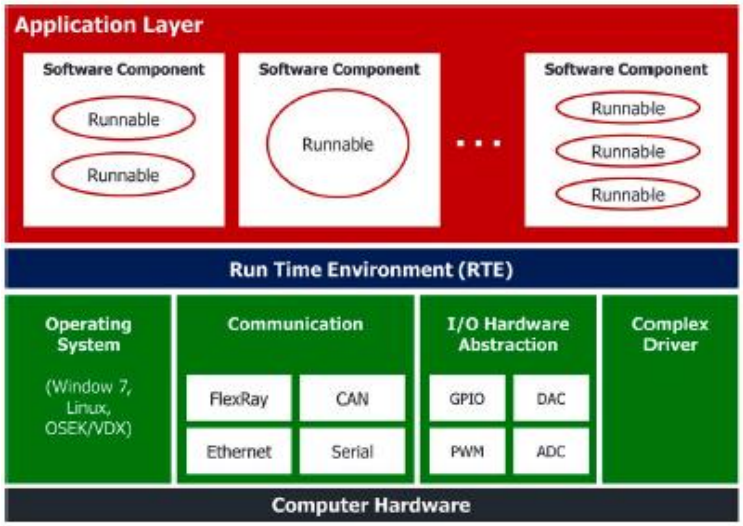

Fig 2. Software Architecture [22]

Out of the six issues identified, three fall under the *functional architecture* perspective: the "functional distribution" section and "data flow" section discusses the division of responsibilities into functional components and information exchange in between them, while the "fault management" [1] section discusses a specific functional components worthy of special consideration. Another two issues fall under the *software architecture* perspective: the "software platform" section and "data model/interprocess communication" section respectively discuss the overall software architecture and data & communication schemes. The sixth issue, which is outside of the scope of either architecture is the "computation platform" section which discusses the hardware selection.

Analysis Framework

To analyze the quality of a piece of architecture design, we follow the guidelines given in the book *Software Architecture in Practice* by Len Bass, Paul Clements and Rick Kazman at the Software Engineering Institute. The book proposes that the most important evaluation is not the functionality itself, though functionality does provide a basis for good design, but the "quality attributes": "systems are frequently redesigned not because they are functionally deficient – the replacements are often functionally identical – but because they are difficult to maintain, port, or scale…". These "quality attribute" are "measurable or testable property of a system that is used to indicate how well the system satisfies the needs of its stakeholders" [7] such as:
Availability: how often the service is available for use, by preventing or handling the faults effectively [7]. In the context of AV, this includes how the system handles module failures and communication failures.

Interoperability: how well the system handles expected and unexpected inter-subsystem exchange requests [7]. In the context of AV, this mainly pertains to whether unforeseen modules can be added onto existing communication networks later in the development stage, or even let new modules join at runtime.

Modifiability: how easily the system adapts to new functions, quality, capacity or technology [7]. In the context of AV, this includes whether the component sizes are manageably small for quick modifications, and how well components are decoupled.

Performance: whether the system responds in a timely fashion, measured by throughput and latency [7]. In the context of AV, the most commonly used metrics include "images per second" for vision-processing and end-to-end response time as measured from perception to actuation [21].

Security: how well the system can counter attacks that aim to compromise "confidentiality", "integrity" and "availability" [7]. In the context of AV, this concerns the separation between functionalities with different levels of real-time [24] and access-restriction requirements.

Testability: how easily the developers can test the system by "injecting faults", "probing states" and "run in sandbox" [7]. In the context of AV, this could be the ability to re-create past runs with different parameters [6], to view the logs in human-readable format, to test individual modules separately, and to test "in the cloud" without a physical hardware [1].

In the following sections, these quality attributes will be major factors of consideration as we assess and compare the architectures proposed by different papers.

Functional Distribution

Identification of Functional Components

Most works in the overall AV literature currently identifies three high-level functional components for the AV system: perception, planning and control [1, 21]. This three-part plan emerges from the general categorization used in the field of robotics [43].

Perception extracts environmental information as well as ego-vehicle status from sensors such as cameras, LiDAR, GPS and IMU. Its submodules can include ego-vehicle localization [1, 20, 45], object detection [1, 20, 45], object tracking [20, 21], semantic understanding (more focus on drawing boundaries on drivable areas than object detection) [1, 20] and world model [1, 45].

Planning makes decisions about behaviors and trajectories based on top-level goal as well as the perceived environment. Its submodules can include mission planning (route from current location to destination) [21], behavior planning (decide among operation modes such as "school zone", "crosswalk" and "parking") [1, 23] and trajectory generation [1, 45].

Control refers to the ability to actually execute the planned motion by breaking it down to duration, strength and direction of acceleration and implementing them using the vehicle's actuators [1, 21]. Its submodules can include longitudinal control (propulsion and braking) [1, 45], lateral control (steering) [1, 45] and reactive control (collision avoidance and emergency braking) [1].

However, many surveyed papers that specifically examine the functional architecture design sought to rethink the three-part structure, and proposed architectures with differing emphases.

Separated Safety System

Jo et al. (2014) advocates for a five-part component design rather than three-part – perception, localization, planning, control, and system management [22]. The separation of localization from perception was not well-explained for, but it is worth noticing how this five-part scheme places more emphasis on system management, which includes fault management, logging facility and the "human-machine interface" (HMI). Design by Behere et al. (2016) represents the other side of such choice: they also recognize the fault management functionality, but they only recognize it as a subcomponent of decision & control. However, Behere et al. do admit that the fault management system actually runs orthogonally and throughout all the other components, including both perception and planning components, but chose not to separate it out.

Serban et al. (2018) coincided with Jo et al. in their decision to make the separation, advocating for "splitting the control system from the safety operations", and offered two arguments [45]. First, Serban et al. argue that it is an instantiation of the "separated safety pattern" proposed by Rauhamäki et al. [52]. The safety system is usually designed for hazardous conditions, and often as a last resort to handle faults beyond the capabilities of control modules. Therefore, across industries, it is generally required that the safety system be developed according to a stricter standard, which could mean using smaller instruction sets or more costly hardware [52]. It is therefore good practice in architecture design to develop the safety system separately, avoiding the cost and restrictions associated with making the entire control system compliant to the stricter safety code [52].

Second, Serban et al. (2018) argues that as the autonomy level gradually goes above 3 in the AV industry, OEMs will need to expand the safety system's responsibilities beyond just handling software failure, but also include handling extraneous environment interactions that the control module fail to handle [45]. As the vehicle assumes full responsibility for decisions in all scenarios, AV will need dedicated "safety reasoning" modules to reason about "self-sacrifice", and such modules will likely be standardized and mandatory [45, 53].

When put under the analysis framework introduced at the beginning of the section, the separation design would increase modifiability by decoupling two increasingly divergent functionalities.

World Model

Besides separating out the safety system, Serban et al. also differs from the three-part component design in that it re-divides functionalities that fall under the scope of perception, planning and control [45]. Serban et al. divide these functionalities into three different categories: sensor processing, world modelling and behavior generation. The sensor processing component corresponds to perception. The behavior generation component encompasses both planning and control. The world modelling component serves only as "a buffer between sensor processing and behavior generation". The world model contains both current and historic "knowledge about images, maps, entities and events, but also relationships between them". More importantly, it provides an easy database interface to downstream behavior generation modules, with query and filter APIs [45].

Similar to Serban et al., the architecture proposed by Behere et al. (2016) also features a "world model". According to Behere et al., The world model is either dynamic or static. A static world model is usually implemented as a "layered-map", with more permanent features such as roads and traffic lights in the background layers, and more temporary features such as pedestrians in the foreground layers [1]. A dynamic model stores kinetic models of the objects in addition to the information in the static model [1]. Either way, the world model would offer "query, add, remove, concurrency control and replication" functionalities similar to a database [1].

Note that the world model component should not be confused with the persistent storage facility, which could also store knowledge acquired from the perception modules, but for a different purpose (i.e. logging, auditing). Serban et al. separates out the persistent storage to another "data management" component, and stresses that the world model is for real-time access by the behavior generation modules [45].

The real-time nature of the world model is important because it makes the world model a part of the run-time communication scheme. Among the papers surveyed, only Goebl and Färber (2007) offer an implementation of such database for real-time communication in AV [24]. The team decided to explore such database because it recognized that there are latency gaps and temporal resolution differences in between the "processing levels" from raw data to module outputs. Therefore, a "buffer" as mentioned in [45] is needed. This need of a buffer is ignored by other surveyed implementations [6, 22, 44]. Their interprocess communication manage send and receive functionalities with well-defined rules, but manage the latencies in an ad hoc way (more detail in the "data model/interprocess communication section).

Therefore, as robustness requirements increase in the future, the architecture design that incorporates a world model – either as a separate high-level component as in [45], or a subcomponent of perception as in [1] – is more likely to offer more stable performance in terms of latency.

Mapping Functional Components to Computing Units

After the functional components are identified, the next architecture concern we consider is how these components should be mapped to "units of execution" [7]. In particular, we consider how these components are mapped to physical computing units, because such mapping has direct influence over how OEMs's component suppliers package functionalities into individual products.

Centralization vs. Distributed

The literature disagrees over whether AV components should be implemented on a centralized or distributed architecture. Here, we will review the main arguments for each.

A major proponent of the distributed design is the team that built the winning vehicle, *A1*, in the 2012 Autonomous Vehicle Competition held in Korea [22, 23]. The team started with a centralized design, but quickly ran into bottlenecks and migrated to a distributed design when the requirements of the competition expanded and the central computer's performance deteriorated [23]. The team then made comparisons of the two designs based on this experience. According to [22], the main advantages of a centralized design include:
1) Minimal network complexity as all the information flows to the same destination, and therefore easy information sharing [22]
2) Minimal delay in communication, because there is little pre-processing happening outside the central computing unit [22]
Besides, other sources of analysis identified two additional advantages of centralized design, specifically when used for the perception component:
3) A central machine with direct access to all the raw data has greater flexibility to change and upgrade its algorithms [54], because it doesn't build on any assumption about the filtered format of any particular pre-processing module.
4) The fusion algorithm located at the central machine can assume conditional independence for all the incoming data (useful for Bayesian fusion algorithms such as Kalman filter and particle filter), because there is no intelligence in the leaf modules and therefore no need/possibility of communication between modules prior to reporting to the central machine. [55]

On the other hand, Jo et al. (2014) identified the following advantages of distributed design:
1) More space to increase "computational complexity" due to stronger parallel computing capabilities. A distributed system with computing units dedicated to small sets of tasks requires much simpler scheduling algorithm and provides stronger real-time guarantee than a centralized machine with multi-core or multi-CPU [22]. The greater execution time stability was also demonstrated quantitatively through experiment [23]. The following graph shows increased average completion time but reduced tail latency with distributed design (Lower)

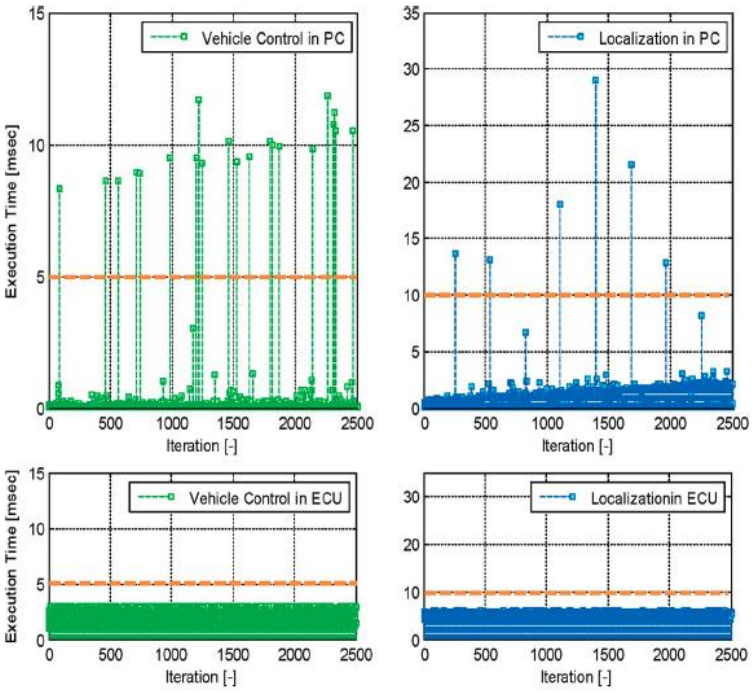

Fig 3. Execution on Centralized vs Distributed Design [23]

2) Greater fault tolerance, as physically disjoint units could back each other up [22, 54]. However, the distributed design also requires a more complex health monitoring system to fully enjoy this benefit, because it carries a higher risk of partial failure [6].
3) Computing units could be placed physically closer to the sensors/actuators they serve, thus reducing wire lengths (weight & cost) and noise at the pre-processing stage [22, 23].
4) Parallel development and testing. The distributed design reduces the scope of consideration for sub-teams responsible for each module, as there is minimal spill-over effect when changing configurations or resource requirements [22, 23].

After weighing the pros and cons of the centralized vs. distributed design, Goebl and Färber (2007) [24] reached a different conclusion from Jo et al. (2014, 2015) [22, 23]. In an effort to build an accessible research platform, Goebl and Färber prefer a centralized system because it is easier to replicate the hardware with "commercial-off-the-shelf (COTS) components" across research labs [24]. More importantly, Geobl et al. point out that the distributed design could lead to "an early partition of software modules on different PC, prohibiting a later rearrangement of modules and shifting cooperation from software to less flexible hardware interfaces" [24]. Jo et al. also recognize these two problems with the distributed architecture and proposed solution for each. For the hardware replication problem, Jo et al. propose using standardized "software platform" to unify development environment despite different hardware choices [22]. This is definitely viable and such standardization is currently under progress with the AUTOSAR Adaptive Platform [26]. However, compliance to the standards still require more efforts from the developer than simply purchasing the same hardware and OS installations off-the-shelf. For the "early partition" problem, Jo et al. propose a centralized "Software Component Design" step before the sub-teams head off to work on each component parallelly [22]. Such a work flow would be more realistic with a more standardized architecture, but less useful in a research project that anticipates large structural changes.

In conclusion, the debate over centralized vs. distributed architecture is far from settled [54, 56], with diverging views from both the academic research community [6, 22, 23, 24] and the industry [54, 55]. However, the debate centers around identifiable issues, including communication speed and noise, computational complexity, fault tolerance, collaboration accessibility, and modifiability. Which design wins out will likely depend on how well each addresses these issues in the coming years.

Decoupling the " Vehicle Platform"

Though fully autonomous driving has not entered people's life, an increasing percentage of cars are not equipped with partial autonomous features called "Advanced Driver-Assistance System" (ADAS). These ADAS systems provide functionalities such as Lane Keep Assist and Automatic Emergency Braking System, and are available on 92.7% of U.S. cars in the market as of 2019 [57]. To minimize re-design cost, it is therefore in the interest of OEMs to pursue an AV architecture that allows maximum reuse in the transition from partially autonomous cars to fully autonomous cars. Besides cost savings, OEM would also prefer designs that decrease product-line risk, by allowing easy degradation back to non-autonomous design if the fully autonomous route proves infeasible [1].

Motivated by this stakeholder need, Behere et al. (2016) believe it is in the industry's interest to pursue a clearer decoupling of "the vehicle platform" from the "cognitive intelligence system", so that the automakers could simply plug-on the cognitive layer when needed [1]. The cognitive intelligence only need minimal knowledge of the vehicle's physical properties, because it doesn't actually "drive" the vehicle – it simply issues high-level "motion requests" (time series of vectors describing the desired trajectories) to the vehicle platform [1]. The trajectory execution functionality, which needs intimate knowledge of the vehicle and traditionally has been part of the AV intelligence, should be entirely offloaded to the vehicle platform [1]. (typo in the cited graph b's upper "Trajectory execution" box, which should be "Trajectory generation")

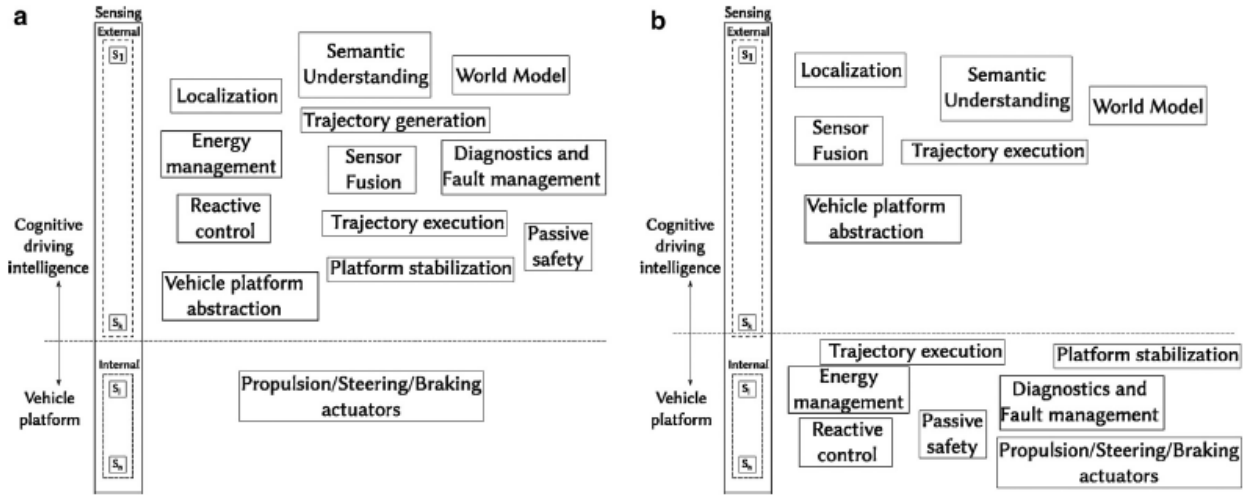

Fig 4. Offloading Trajectory Execution to the "Vehicle Platform" [1]

In this way, the only required knowledge for generating these commands are simple static variables such as actuation latencies, which can be easily loaded on demand [1]. In this way, it is extremely easy to turn a non-autonomous vehicle into an autonomous one and versus versa [1]. Note that this model does places a heavier burden on the base vehicle by assuming that the vehicle platform is smart enough to execute the requested motion, but this should not be too challenging given that what motivated this stakeholder need is the increasingly advanced semi-autonomous features in

modern cars, such as ADASes, which already expose longitudinal and lateral controls to software control [1].

Behere et al. also point out that since the vehicle platform is usually developed by embedded system programmers who use real-time operating systems (RTOS), while the cognitive intelligence system is usually developed by computer scientists who use resource-heavy general-purpose operating systems (GPOS), such decoupling would also facilitate parallel development using each groups' preferred platform [1].

This facilitation effect is evidenced by Jo et al.'s (2015) computing unit mapping choice in their implementation of A1 [23]. Jo et al. did not follow Behere et al.'s architecture design, but they nevertheless concluded through experiment that the ideal allocation is to separate out the actuation modules (longitudinal and laternal control) into one RCP-ECU (Rapid Prototyping Electronic Computing Unit), the cognitive intelligence modules (sensor fusion, planning, vision) into another two industrial computers (Intel Core 2Duo with Windows 7), and the rest of modules into 13 other ECUs [23]. (graph below). The choice was motivated by the actuation module's stricter real-time performance requirements, and easy access to cognitive libraries (OpenCV and BOOST) on GPOSes such as Windows [23].

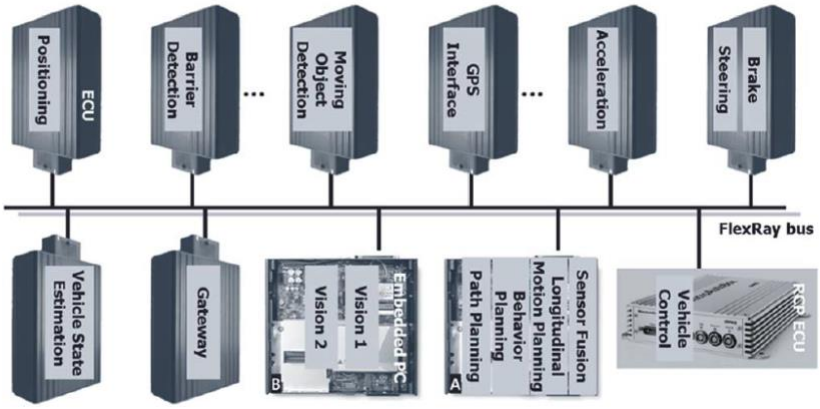

Fig 5. Distribution of Functional Components over Multiple Hardware Units for A1 [23]

In conclusion, the decoupling of the vehicle platform seems like a logical step towards greater AV-software modifiability. Since the actuations and cognitive modules have different requirements for computing hardware anyway, it is a relatively low-risk decision. If such decoupling can be soon standardized through a well-designed interface, such as the motion-request vector interface proposed by Behere et al., it would help component suppliers focus on iteration of either the vehicle platform or the cognitive system without worrying too much about influencing the design of the other, thus increasing modifiability.

Functional Redundancy

Full redundancy is sometimes implemented to act as a "catch-all" defense against system failures [1, 45], such as in the newest model of Tesla Full Self-driving Chip with duplication of all the computing units as well as data input and power supply [58]. However, in the AV architecture, redundancy is not just about fault handling. It is also an integral part of sensor plan design – redundancy is often used to improve accuracy when operating under normal condition. However,

the debate over the optimal level of redundancy has become a source of diverging architecture designs.

The need for sensor redundancy arises from the complementary nature of the advantages and drawbacks of various sensors:
- The GPS/IMU system couples the relatively infrequent (10Hz), accurate GPS updates and relatively frequent (200Hz), inaccurate inertia measurements to provide localization. The system is fairly accurate most of the time but can be dysfunctional when GPS signal is obstructed by tunnels [20].
- The LiDAR system offers high accuracy, and therefore can be used to localize against high-definition maps, but LiDAR is susceptible to noise caused by rain and dust [46].
- The vision system (camera) provides color information that LiDAR cannot provide, but performs less consistently "across different illumination conditions" than LiDAR [2].
- Sonar and radar can be used as the "last-line of defense in obstacle avoidance" due to their superior performance in "short distance detection", but helps little with object detection and tracking [20, 44].

Therefore, AVs are generally equipped with several of these sensors, and use the fusion of their outputs for localization, object detection and tracking. Since the GPS/IMU system and short-distance radar are relatively cheap and already prevalent in non-autonomous cars, they are generally included in the sensor plan. The focus of dispute is therefore whether vision is enough for high-precision perception, or if LiDAR should also be included in the fusion.

On one side, LiDAR and vision do complement each other in various illumination and weather conditions, increasing the system's availability, and "many successful implementations of autonomous driving rely heavily on LiDAR" [46], such as those by Nvidia/Audi and Waymo [21]. This LiDAR/vision fusion process could either takes place at the "feature level", corresponding the pixels in the vision image to points detected by LiDAR, or at "decision level", combining the resultant probabilities parallelly calculated from vision and LiDAR systems at late stage [2]. It is also possible to design multi-level fusion systems (graph below) that performs fusion in hierarchical layers, each taking both sensor data and the previous layer as inputs for even better robustness [59, 61]. Therefore, there is high variability in architecture even within the realm of LiDAR/vision-fusion design.

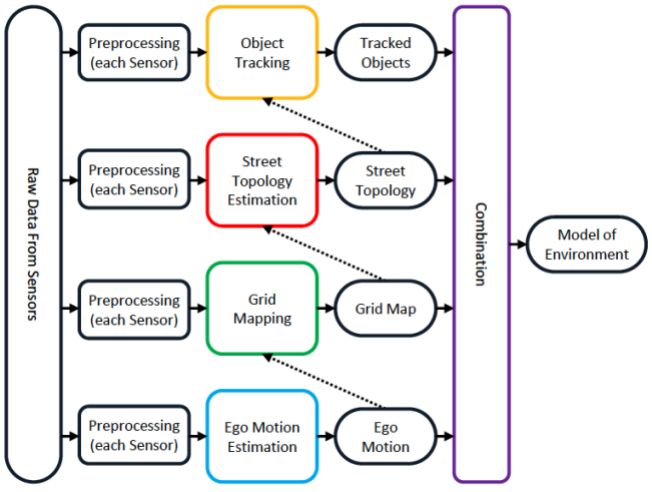

Fig 6. Hierarchical Fusion Scheme [61]

On the other hand, due to the extreme price-tag on LiDAR system ($75,000 per unit [60]), there have also been many AV companies that focus on vision-based systems, such as Mobileye, Tesla and Mercedes Benz's Class S 500 [21, 44]. The vision-based design is also touted for its higher computational efficiency in localization tasks, because its visual odometry approach consists of "highly parallel data-processing stages" and also makes heavy use of easily accelerated vector computations, while the LiDAR-based design relies on the sequential Iterative Closest Point algorithm [46]. Besides, the reduced availability of high-precision data caused by the lack of LiDAR could be partially compensated by software innovations [44]. Zong et al. (2018) propose an "environment mapping loop" (graph below). that uses the "rotation and transmission matrix" derived from historic frames, in combination with the current detection to cover blind spots in the cheap sensor set: with the assumption of constant acceleration between consecutive samples, predicted but undetected vehicle position proposals are assumed to be "either out of range or misdetected", so they are overlaid on top of current detections in the form of "probability ellipse" with decreasing values from center to boundary [44]. This software innovation helps the vehicle achieve a tracking accuracy of "mostly under 10 cm and no more than 20cm deviation" on a 23km test route, using only stereo cameras, radars, and sonars as sensors [44].

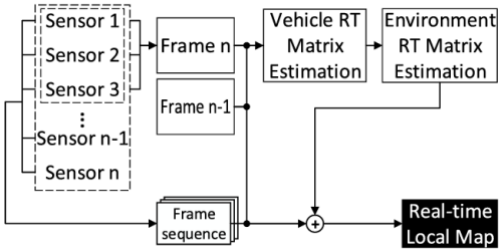

Fig 7. "Environment Mapping Loop" [44]

In conclusion, since major AV competitors disagree over the optimal sensor plan, architecture designers have to reserve considerable flexibility for sensor changes. Even with the same sensor plan, the fusion component could be placed at various junctures of the perception pipeline, and could be centralized ("feature level") or decentralized ("decision level") [2]. Besides, as shown by the "environment mapping loop" design [46], to cope with data availability, sensor changes also have rippling effect on the software complexity of downstream modules. The uncertainty in perception sensor selection is therefore a major roadblock to building a standardized architecture in the industry.

Data Flow

In the previous section, we identified functional components and discussed their organization – distributed vs. centralized, with or without redundancy – essentially, we are drawing the boxes in the functional architecture (refer to graph in "Discussion Organization" section). In this section, we discuss how these components collaborate, drawing the arrows in the functional architecture. Since most of the surveyed architectures are distinct, it is not very meaningful to compare every data flow one-to-one, as some may not exist or make sense in alternative architecture. The following data flows are identified because they were discussed by multiple sources for their design benefits.

Perception to Trajectory Generation

The trajectory generation module is responsible for generating obstacle free trajectories represented in coordinates, while keeping in mind energy and directional constraints [1]. It is the often the last module in the perception-planning pipeline, feeding directly into control. Therefore, sensor data usually takes a long journey before reaching trajectory generation. However, researchers have found it beneficial to include a short-cut data flow directly between sensor outputs and trajectory generation.

The first argument for such a bypass is speed. When a fast-moving obstacle poses an impeding threat, there simply isn't enough time for the new information to be processed and passed down through the usual pipeline [20]. In some designs, this problem is alternatively solved by adding a full redundant system ("reactive control") that separately handles this kind of situation. This extra system monitors a few selected sensors, has much simpler logic, and therefore takes much less time to process. In this alternative design, instead of feeding the sensor information to the general pipeline's trajectory generation, control signals are directly issued to control modules and override those from the main logic pipeline [1].

The second benefit is computational efficiency. Wei et al. point out that a major problem they identified when refining the AV planning framework is that the motion planner needs to simultaneously consider "road geometry, vehicle dynamics, surrounding moving objects and static obstacles", which greatly increases the complexity of the planner problem [61]. Wei et al. propose a "novel planning framework" that feeds the perception information to downstream modules to re-evaluate the path consider moving obstacles, so that the high-level planner solves only the static problem [61]. Liu et al. (2018) also state a similar idea when analyzing such data flow, that it would "serve as a backup for the traffic prediction" [20].

With regard to the analysis framework, the perception to trajectory generation data flow increases the system's availability to handle extraneous road conditions.

Feedback from Actuations

The control modules themselves are often based on the principle of feedback -- the control modules monitor the completion of past commands and issue new commands in view of the past commands' completion [20]. However, not all architectures allow actuation feedback to further bubble up to trajectory generation and behavioral planning modules. Tas et al. (2016) identified this as one of the most important corrections that an architecture should make to increase robustness [43].

The feedback mechanism has two benefits. Firstly, it provides instant update to the planning modules in case the controllers fail to execute the generated trajectory and cause significant deviations from the expected position. A direct feedback to the planners would take much less time than waiting for the perception modules to detect it, and in case of extreme environment conditions, the perception sensors themselves may not be able to provide reliable information. The planner module could then rely on a fallback trajectory to guide the vehicle to safety – but it is only useful if feedback from controllers is timely [43]. Secondly, the feedback could be used to let the planning modules familiarize themselves with the execution pattern of the controllers. This is especially important for an architecture with decoupled vehicle platform – the mounted cognitive intelligence system may not assume perfect execution of its plans, and should "learn and adjust the model parameters" about the vehicle platform from the feedback [1].

The "Tee-and-Join" Model

Though specific data flow choices are heavily dependent on the algorithms used and therefore vary from architecture to architecture, there have been efforts to identify generalized patterns that could be universally useful for organizing the data flows on a high level.

Serban et al. (2018) propose using the "tee-and-join" model to represent the dataflow [45] (graph below). The model assumes that outside information enters through the "sensors abstraction" module at the lowest level of hierarchy, and then from each "pipeline" (modules on the left-hand side), either travels up to the pipeline with a higher level of abstraction, or travels sideway to the corresponding "control loop" (modules on the right-hand side) of the same level of abstraction [45].

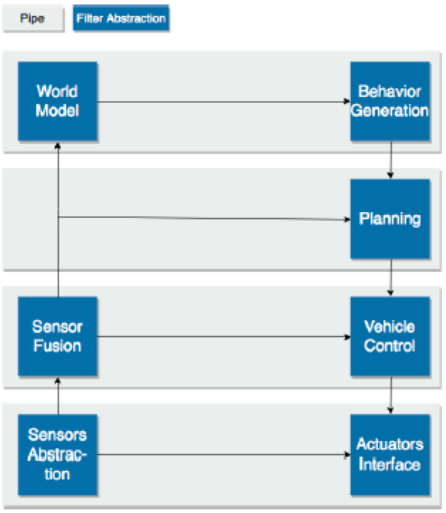

Fig 8. "Tee-and-join" Model [45]

Though Serban et al. did not present an implementation of their design, one surveyed paper by Goebl and Färber (2007) did come up an implementation that closely resemble this design [24] (graph below). Goebl and Färber based their design off an early work by Maurer and Dickmanns [63]. Maurer and Dickmanns' data flow organization echoes Serban et al.'s in that the architecture is structured hierarchically according to the data's level of abstraction – the lower levels process data is represented by "differential equations, state space representation, recursive estimation or controllers and filters in the frequency domain", while the higher levels' data use "computer science representations" such as "boolean logic, decision trees and automata" [63].

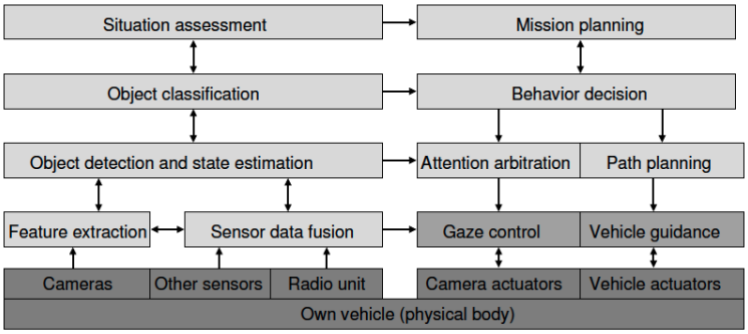

Fig 9. An Implementation of the "Tee-and-join" Model [24]

The vertical data flows up to higher level of abstraction are intuitive to understand, as one of the important goals of the AV software's goal is to represent and reason with the road condition abstractly, so that the system could also deal with unknown situations [63]. The horizontal dataflows are less obvious. Maurer and Dickmann explain the horizontal dataflows by pointing out that many control modules only need input on its same level of abstraction and no deeper insight – the brake pressure controller (bottom level actuator) need no further information than the brake pressure (bottom level sensor), and the state controllers (middle level control) in lane-following mode need no further information than drive area detection and object detection (middle level cognition) [63]. Mauerer and Dickmanns also believe that this rough structure should be applicable to most AV systems regardless of their tasks [63].

Data Model/Interprocess Communication

In the previous section, we examined the data flow between functional components, focusing more on the question of what data is being sent rather than how it is being sent. This section will focuses on the interprocess communication system (IPC) [6] from the perspective of "software architecture", i.e. how the data is being sent. The purpose of the IPC is to abstract the data transmission infrastructure from the module programmers, by taking care of IO threads, serialization and addressing in an efficient, reliable manner. Three designs from different researches are examined and compared.

As will be shown, these data models are still prototypes and there is quite a lot of uncertainty as to which standard the AV industry will converge to. However, the industry does have established technology on a lower level – the vehicle bus systems that support these data models. The vehicle bus systems are standardized technologies that had been evolving since the 1990s. After the "architecture" topic, we will dive into those vehicle bus systems, and explore how their properties shape the data models.

Distributed Pub-sub Model

McNaughton et al. (2008) designed this IPC scheme ("SimpleComms") to maximize flexibility [6]. To make partial failures safer, the IPC is pub-sub based, so that when a sender or receiver node fails, the other end wouldn't have to wait for the reply. The communication is also completely anonymous, and pub-sub channels could be created anytime, thus offering greater interoperability and allowing new nodes to join even at run-time [6].

Among all its features, this IPC's scalability may be its most commendable innovation. Each machine in the network hosts a "scs" (SimpleComms server) which serves as a hub handling all the incoming and outgoing message for the multiple modules running on this machine [6]. If a message is intended for multiple modules on the same machine, it will only be sent over the inter-machine TCP connection once, and the scs will subsequently relay it to multiple destination modules [6]. This multi-hub distributed design eases the inter-machine bandwidth requirement as the system scales up [6].

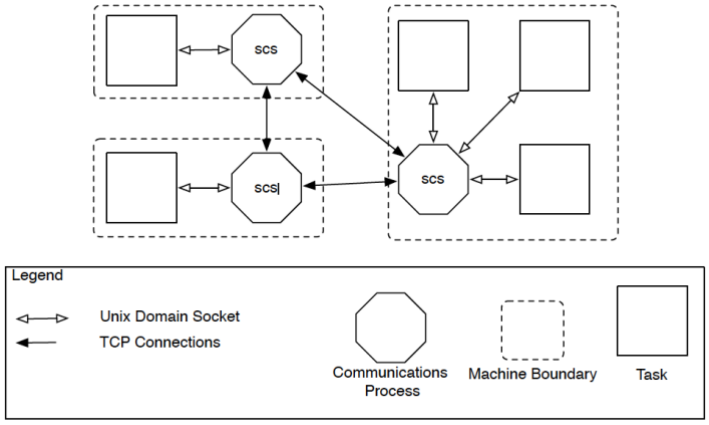

Fig 10. SimpleComms' Distributed Server Scheme [6]

McNaughton et al. also designed the IPC to be "stateless" to further increase its flexibility, meaning that each message contains all the information one needs to understand it [6]. This feature was implemented so that the restarted nodes could catch up the conversation immediately [6], increasing the system's availability. However, it also makes the system vulnerable to security attack, as it only requires a brief period of security compromise to get a full picture of the current state.

Single Server Destination Based Model

Zong et al. (2018) also designed their IPC ("Cocktail") to decouple nodes, so that the system offers greater availability in case of partial failure [44]. However, instead of a channel based pub-sub model, a destination based model is used. To enable nodes decoupling in a destination based model, Zong et al. employed the UDP instead of TCP, so that packages destined for disconnected modules are simply dropped [44]. The connection checking is performed by the virtual server that connects to all the nodes and mediates all the messages [44].

Algorithm 1 Project Cocktail Server Logic

Input: Packets from Modules
Output: Packets to Modules

```
Listen for every module m in parallel
while module m received packet P_m do
  if P_m is Connection and m is allowed to be connected
  then
    set TargetAddress[m] to the IP:port from P_m
  else if P_m is DisConnection and TargetAddress[ m ]is
  IP:port from P_m then
    set TargetAddress[m] to NIL
  else if P_m is DataUpdate then
    if P_m is SinglePacketData then
      Send P_m to TargetAddress[Target specified in P_m]
    else if P_m is PartialPacketData then
      Send P_m to TargetAddress[Target specified in P_m]
    end if
  end if
end while
```

Fig 11. Cocktail's Server-side Code, with Connection Checking [44]

Testing on this single server design reveals its inferiors scalability compared to the SimpleComms' [6] distributed design. Controlling for the net amount of data transferred, the "all-to-all" mode experiences greater latency than the "single-to-single" mode, suggesting that "repeating listening to one module should be avoided as much as possible" [44]. In SimpleComms, the message relaying workload is distributed across the local servers [6], thus building in crowding avoidance for each individual module.

Central Database Model

Neither of the previous two models manages backlogging systematically. SimpleComms keep an incoming and an outgoing queue for each module. Each queue is of a fixed length, so that new messages exceeding the count limit will be dropped [6]. This choice is clearly suboptimal as it keeps the stale value rather than the latest more relevant value in case of delay. Cocktail keeps the departing and arriving messages in the "Packet Base", but did not specify how backlogging is handled [44].

As mentioned in the "world model" section, Goebl and Färber (2007) recognize the prevalence of backlogging in AV system, caused by the wide range of "temporal resolution" and cognitive complexity of each processing level [24]. Therefore, Goebl and Färber designed a database style IPC ("KogMo-RTDB"), where all stored objects are required to declare their frequency, back history length and maximum message length, so that other modules could explicitly expect and plan for when and how frequently the objects should be polled [24]. Besides, the buffer is circular so that the latest information is kept [24].

Though the central database design helps enforce object definition standards and clock synchronization, it also introduces locking delays when many modules attempt to access the central database at the same time [24]. Goebl and Färber partially mitigated this delay by building the database library on top of a real-time framework, Xenomai, which couples with a Linux kernel [24]. In the future, data model designs could potentially combine the advantages of SimpleComms and KogMo-RTDB by building a distributed database style IPC that manages backlogging systematically while spreading the access workload across multiple machines.

Software Platform

In this section, we backtrack to look at the bigger picture of the software architecture (defined at the "overview" section), in which the IPC systems we examined in the previous section is a part of. The main purpose of defining a standardized AV software platform is to increase modifiability: to allow the application modules (discussed in the "functional component" section) to be developed and modified free of hardware consideration, which can be a major headache due to the heterogeneity of computing hardware (see "computing platform" section) used in a distributed AV system [22].

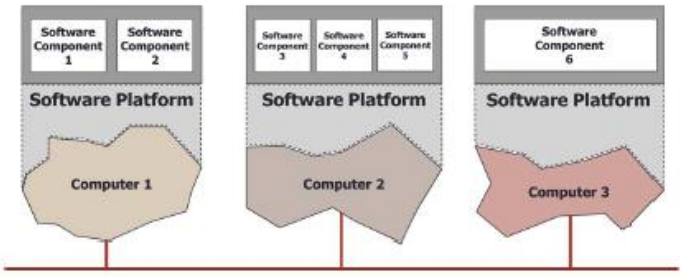

Fig 12. Software Platform Makes Software Hardware-agnostic [22]

Most of the surveyed literature seem to agree on the three-layer structure of software platform, which are the OS layer [44] (Basic Software [22]), the data transmission layer [44] (RunTime Environment Layer [22]), and algorithm layer [44] (application layer [22]) from bottom to top. The OS layer are closely coupled with the hardware implementation (i.e. computing platform, vehicle bus system etc.) and therefore has highly variable implementations [22]. To increase interoperability of the application modules, the specific network, operating system, and I/O interface of the OS layer must be abstracted away, and this work is done by the RTE layer [22]. Since AV sits at the intersection of the automotive field and the robotics field, both fields has offered an RTE solution.

Robotic Operating System

The Robotic Operating System (ROS) is an RTE solution borrowed from the robotics field. It is not an operating system as its name suggests [64]. It is a middleware providing functionalities including "hardware abstraction" [64], "device drivers" [64], and pub-sub style IPC [47]. ROS is designed to work on a heterogeneous hardware cluster and make them work as "a single entity", [34] which is exactly what the RTE layer is expected to do. Besides, ROS is also highly modular, as each module can be encapsulated in a single ROS node and added/removed with minimal impact on other ROS nodes [47]. However, ROS's lack of network security [34] and real-time performance [47] makes it more suitable for research and experiment than commercialization. Besides, though ROS offers convenience as a rich open source platform, its design is not optimized for AV applications' requirements. For example, the Cocktail IPC examined in the "data model" section exhibits superior real-time performance than ROS's IPC on both latency and throughput when used in an AV system [44].

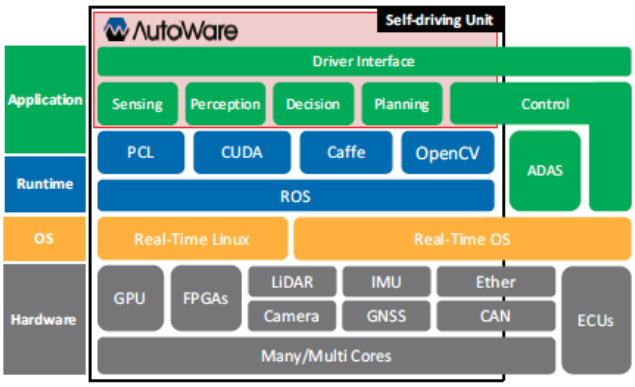

Fig 13. Structure of Software Platform of a ROS-based AV Implementation [47]

AUTOSAR Adaptive RTE

The AUTOSAR Classic software platform is time-tested, secure, and widely adopted by the automotive industry. However, its standardized interface only works with embedded operating systems, which are good for real-time applications and are adequate for non-autonomous vehicles [26]. However, the increasingly intelligent AV modules require interface to the more versatile APIs provided by POSIX-based general-purpose operating systems [22]. In 2014, Jo et al. (2014) specifically implemented their own RTE for AV out of this concern [22]. Fortunately, the AUTOSAR consortium also recognized the importance of incorporating POSIX OSes, and released the first version of RTE interface for the new "AUTOSAR Adaptive Platform" in 2017 [26]. No longer bound to operate on an embedded OS, AUTOSAR Adaptive's RTE can schedule modules and allocate memory dynamically, but it will still maintain a high standard for safety and real-time performance appropriate for commercialized automotive [26].

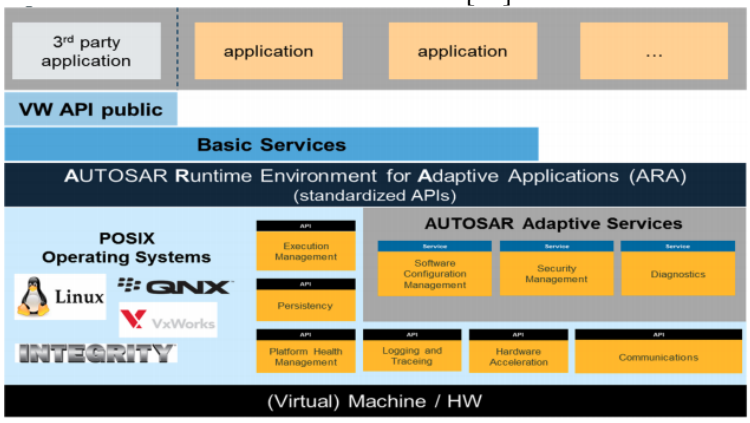

Fig 14. Structure of Software Platform Based on the AUTOSAR Adaptive Standard [65]

Fault Management

As safety is one of the most important factors to consider when designing the AV architecture, almost all surveyed full system designs touch upon "fault management" [1, 46]. Though the most bullet-proof way to ensure availability is to have duplicates of the same system running parallelly at all time, switching back and forth when faults happen, it is certainly not the most efficient way, as duplication aggravates energy, heat and space constraints on the single system [1].

Degradation

In lieu of a duplicate system, many designs incorporate a fault management module that listens to the health status signals of all other modules at all time, and switches the entire system to fault mode ("degradation" [1, 26]) when a particular module fails to update its status [23]. The AUTOSAR standard does not specify behavior for the fault mode, but simple asks the implementation to provide a "well-defined strategy" [26].

Goebl and Färber's (2007) design "initiates an emergency brake maneuver using the last known situation data available in the database" [24]. This is safer than Jo et al.'s (2015) design which pauses the vehicle regardless of location [23]. However, when a fault happens, there is no guarantee that the surviving software is still capable of calculating a safe route from the last known data. Behere et al.'s (2016) design solves this problem by requiring the "cognitive system" to always output two routes to the "vehicle platform", one to destination and one to safety [1]. Therefore, even if a critical fault happens in the route planning module, the "vehicle platform" can navigate to safety by using the latest safety route it received before the fault[1].

Recovery

While degradation is a viable strategy for lower level AV (level 0-3), higher level AV (level 4-5) system has to overcome the fault and continue its driving duty, because humans drivers are not expected to take over as driver at any time [45]. Therefore, the "fall-back mechanism" has to seek recovery rather than degradation [45]. One common approach is to issue "restart" commands to modules that failed to send a health signal to the fault management module for a period of time [6]. In a centralized design, the actual restart could be performed by the fault management module itself [44]. In a distributed design, after the central fault management module issues the command, the actual restart could be delegated to a specialized local process running on each distributed machine [6]. Further, Liu et al. (2017) consider the possibility of a "master" failure, i.e. failure of the fault management module [20]. Liu et al. propose using the ZooKeeper system [66] to keep multiple fault management modules on different machines, so that there is always a replacement master to continue monitoring the system [20].

Computing Platform

As the state-of-art perception algorithms increasingly gravitate towards computationally intensive techniques such as deep learning [2, 20], a high-performing full AV system can no longer rely on a solely CPU-based computing platform to achieve reasonable response time, even if the researcher intentionally design the system to be more CPU-friendly [47]. However, though incorporation of high-horsepower computing units, such as GPUs, can boost performance, it should be done selectively and only after careful consideration. For example, one commercial AV hardware system consists of a 12-core CPU (Intel Xeon E5) and four to eight GPUs (Nvidia K80 GPU) would run with power of up to 3000W [46]. On average, 1000W power raises the in-vehicle temperature by 10°C in a minute [21], meaning that the described hardware system could raise the in-vehicle temperature by 30°C without a cooling system. However, adding a matching cooling system would empirically doubles the power consumption of the whole system, leading to more than 10% reduction in driving range if mounted on a Chevy Bolt, as the vehicle could only carry a limited amount of fuel/electricity [21]. Therefore it is important to tailor the selection of the "acceleration platform" [21] to the particular algorithms used, so that it strikes a fine balance between higher speed and lower power consumption [21].

Types of "Acceleration Platform"

GPU: Graphical Computing Unit. While a CPU typically has no more than 50 cores (a core is "an independent processing unit that reads and executes instructions of a program" [67]), each capable of performing complex instructions, a GPU typically has thousands of them, each capable of performing only simple and repetitive mathematical tasks [68]. Therefore, GPU is especially good at parallelizing the simple yet numerous vector computations in the neural network algorithms [68].

DSP: Digital Signal Processor. DSPs usually have instruction sets especially optimized for digital signal processing algorithms, with each instruction performing task that could take several instructions in general instruction sets [69]. In this way, DSP saves time and energy for its specialized field [69]. In an AV system, DSP could be used for the vision modules, as there are commercial DSP specializing in vision operations [46].

ASIC: Application-Specific Integrated Circuit. ASICs are custom-made chips whose type, number, layout, and internal structure of microprocessors and memory blocks are designed for specific use [70]. The specificity varies depending on the type of ASIC [70], but can be down to an algorithm level, such as the "feature extraction" algorithm within the localization module [21], with the algorithm printed in hardware as logic gates.

FPGA: Field-Programmable Gate Arrays. Similar to ASICs, FPGAs also allow the users to program their algorithms into the hardware using logic gates. However, FPGAs come with versatile logic blocks which can be re-configured to express a chosen logic by user after it is manufactured [71]. In theory, the performance of FPGAs would always be same or worse than ASICs of the same configuration, but FPGAs offer more flexibility in case of design change [70].

Choice of Platform

Lin et al. (2018) compare the performance of GPU, ASIC and FPGA for three computationally intensive algorithms: object detection (neural network), object tracking (neural network), and feature extraction (small vector filters applied all over images) [21]. For object tracking and feature

extraction, Lin et al. found that ASIC performed best in terms of both power consumption and tail latency. For object detection, GPU performed best in terms of tail latency, but FPGA may be the better choice because it consumed much less power [21]. However, though the study concluded in favor of FPGAs over GPUs for all three tasks because of their lower power consumption [21], the study did no control for latency (the FPGAs had worse latency), so it wasn't clear if FPGAs were really more efficient. Nevertheless, the study presented viable solutions: with full ASIC setup and ASIC/FPGA mixed setup, the end-to-end latency was less than the 100ms goal, which is the human driver response time, and driving range reduction was a mere 2.5% [21]. Another study by Liu et al. (2017) compares the performance of GPU and DSP on object detection and feature extraction, and found that GPU performed better using less energy for object detection, while DSP performed better using less energy for feature extraction [46] [GRAPH 16-dsp-gpu].

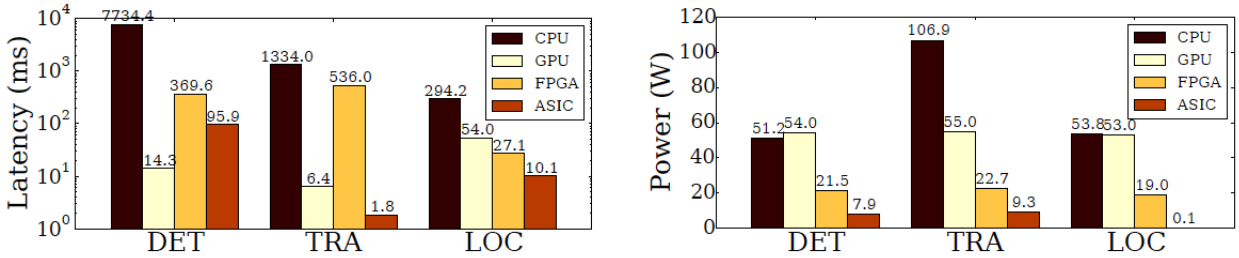

Fig 15. Comparison of Acceleration Platforms' Latency (left) and Power Consumption (right), on Object Detection, Tracking and Feature Extraction Tasks [21]

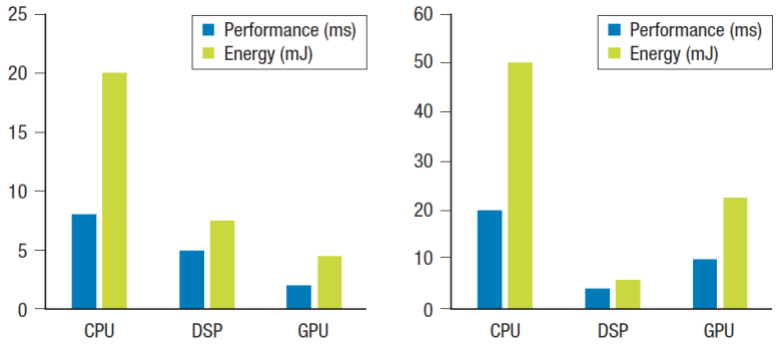

Fig 16. Comparison of Acceleration Platforms' Latency and Power Consumption, on CNN (left) and Feature Extraction (right) Tasks [46]

From the graphs above, we can safely conclude that the optimal choice of computing platform varies significantly for each algorithm, even for algorithms of the same nature (i.e. both detection and tracking use neural network). Therefore, the computing platform of the entire AV system will most likely be highly heterogeneous (software implication of heterogeneity discussed in "software platform" section). Besides, the specific results in the two studies – results of which computing platform is best for which module – may not be highly instructive, as the optimal choice will likely vary when researchers use a different algorithm for the module.

Vehicle Bus Systems

Overview

As suggested by many AV developers [16, 22, 34], the autonomous vehicle should not be understood as one single giant computer, but more appropriately as hundreds of data centers connected by network – the "vehicle bus system" [11]. Therefore, the in-vehicle data network is an important part of the software architecture. It situates at the "basic software layer" of the three-layer-structure, which is the foundation layer of software architecture [22]. Besides, the vehicle bus system's bandwidth, speed and reliability also has significant influence over the functional architecture design, especially the decision on whether and how to distribute functional components across machines.

In the previous sections, we have mainly focused on the choice of data network at the level of *interprocess communications (IPC) toolkit* [8] (or middleware with IPC capabilities), such as SimpleComms [6] and Cocktail [44], the choice of IPC often builds on the choice, or reality, of the lower level vehicle bus networks. By definition, "an interprocess communications toolkit abstracts the transport mechanisms, such as shared memory or network protocols, away from the module developer." [8]. However, IPC designs are often influenced and restricted by aspects of bus system design, such as the bus system's communication model (pub-sub v.s. client-server), delivery predictability, and dynamic flexibility [22]. For example, the Adaptive AUTOSAR RTE's IPC specification, has recently upgraded to include Ethernet as one of its supported bus systems [10]. This is a result of strong push from autonomous vehicle developers, who are eager to move towards service-oriented IPC [27]. Besides capability restriction, the vehicle bus systems poses an additional challenge to the IPC -- the co-existence of a variety of bus systems in the same vehicle requires interoperability, which is another major concern of IPC designers [6, 22].

In this section, we will introduce the four most common vehicle bus systems for in-vehicle communication, LIN, CAN, FlexRay and Ethernet to illustrate what differences the network types make, why the bus system is often a mix of different networks, and why the autonomous vehicle developers care so much about upgrading to the newest vehicle bus system.

The Rise of Vehicle Bus

The need for the ECUs (Electronic Control Unit) to communicate is not unique to autonomous vehicles. For example, regardless of level of autonomy, modern vehicles typically have "Engine Control Unit, Transmission Control Unit (TCU) and Anti-lock Braking System (ABS)" [11]. The transmission unit needs both engine speed information from the engine unit [11] and wheel speed information from the ABS [12] to perform gear shift safely. Therefore, instead of wiring both the Engine Control Unit and the ABS separately to the transmission unit, a vehicle bus network allows all three modules to share information on a common platform and the transmission unit can read all the information it needs from this single platform. This central bus platform not only reduces wiring complexity but also makes it possible to add or remove additional modules by simply adding or removing them from the network without affecting other modules [11].

As one of the earliest vehicle bus protocols, CAN was officially released in 1986 by Intel and Phillips, and it was later further specified by Bosch in 1991. Over time, it has become the one of the most popular communication standards in cars, but it is too expensive and complex to be implemented for every ECU in the vehicle [13, 14 and 15]. Therefore, with the rise of the LIN protocol in the late 1990s (endorsed by a LIN Consortium of BMW, Volkswagen, Audi, Volvo and Mercedes-Benz) [15], it has become common practice to use the LIN among non-critical modules, such as heating and roof, and use the CAN network in both critical modules and also in the backbone network that connects multiple subnetworks. In this hierarchical approach, LIN is often used for "subnetworks to a primary CAN network" [14].

However, with the advent of higher levels (> level 3) of autonomous driving, the bandwidth requirements of in-vehicle networks increased dramatically, for new demands such as large sensor information flow and infotainment streaming [27]. In 2000, the FlexRay Consortium (Daimler Crysler, BMW, Motorola and Philips) was formed [26]. FlexRay offers a least x10 higher bandwidth than CAN and also features a more dynamic time allocation scheme. In very recent years, Ethernet for cars has become popular and was integrated into the AUTOSAR 4.1.1 standard released in 2013 [26]. Like the hierarchical network with a mix of LIN and CAN systems described above, Ethernet is also currently being used with a mix of local lower-bandwidth subsystems (e.g. CAN) in autonomous cars due to its high cost [27].

In the sections below, each type of vehicle bus systems will be introduced with focuses on their topology & hardware, message frame specification, and synchronization & error handling schemes. Performance and robustness comparisons will be made throughout the sections.

LIN

Topology & Hardware

LIN is a broadcast serial network built with a pub-sub model [15, 16]. It comprises of a single master and a variable number of slaves (typically up to 16 including the master, which can also act as a slave when it responds to its own message) [15]. It uses a single wire and therefore only the master can initiate a message, while the slaves can only respond by filling the frame initiated by the master and listen to the signals being communicated between the master and other slaves [16].

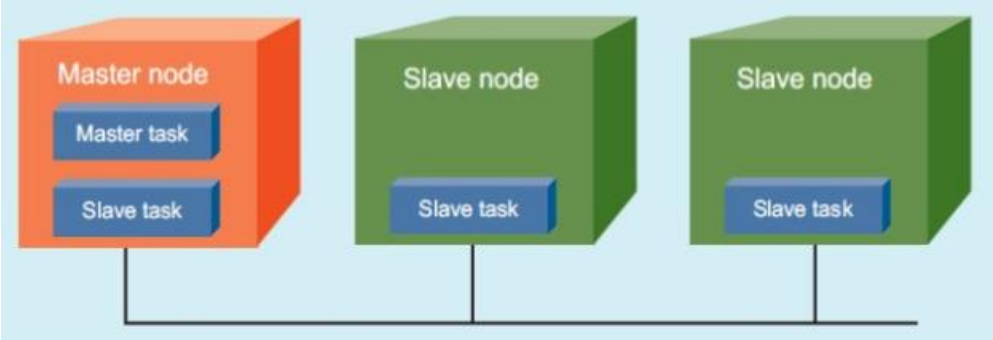

Fig 1. LIN connection. [16]

Message Frame

With the pub-sub model, LIN message frames do not specify destination ID, but only specifies message ID. Every slave node need to listen to all incoming signals, each time deciding whether it needs to publish or subscribe depending on the message ID [16].

The LIN network sends one byte at a time (one dominant bit + 8 data bit + one recessive bit) and each message frame is made up of multiple bytes. As shown in the graph below, the message frame can be divided into a header, which contains synchronization and identifier information, and a response, which contains the message and checksum sent by the responding slave[16].

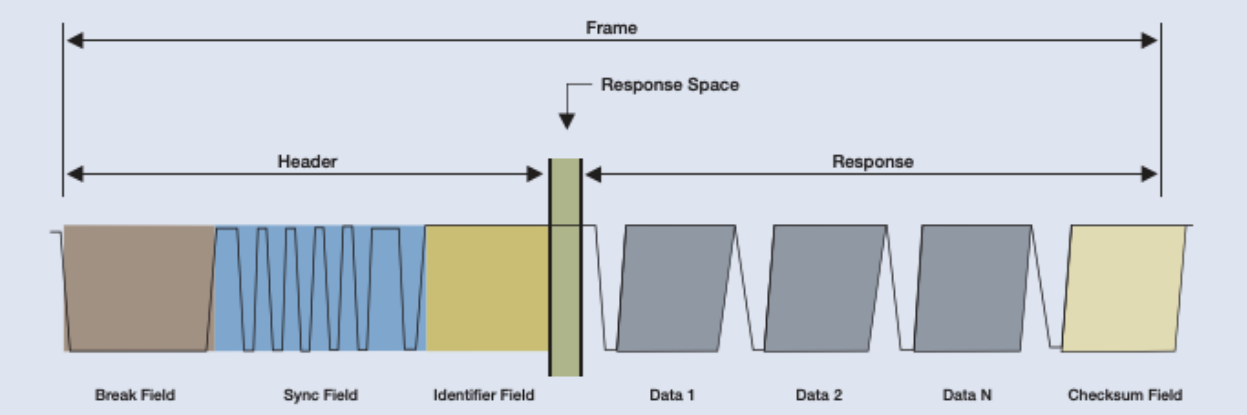

Fig 2. LIN message frame. [19]

Synchronization & Error Handling

As a corollary of the stated coordination method, LIN does not require synchronization crystals in the slave nodes because the master node controls the timing. This is a major cost saving factor compared to CAN [13].

Error detection mechanisms in the LIN protocol include parity check (of the identifier) and checksum (of the message) [16], which are much simpler but also less robust compared to CAN. The master node is responsible for error handling, but LIN does not specify error handling behaviors [16]. Besides, the master node is also responsible for occasional collision resolution, because collisions can arise when an event-triggered frame (rare) is initiated by the master. An event-triggered frame is sent when the master node wants to poll a rare occurrence event among the slaves and therefore can trigger none or multiple responses. However, most frames sent by the master are not of this nature and target a specific slave for response [15].

CAN

Topology & Hardware

There are two major similarities between LIN and CAN. First, CAN is also a broadcast serial network built with a pub-sub model [17]. Second, CAN also uses message ID rather than destination ID in its message frame [16]. However, CAN is much less rigid because it allows multiple masters. Since there are multiple masters that can initiate messages, CAN protocol arbitrates transmission collisions based on the messages' priority level.

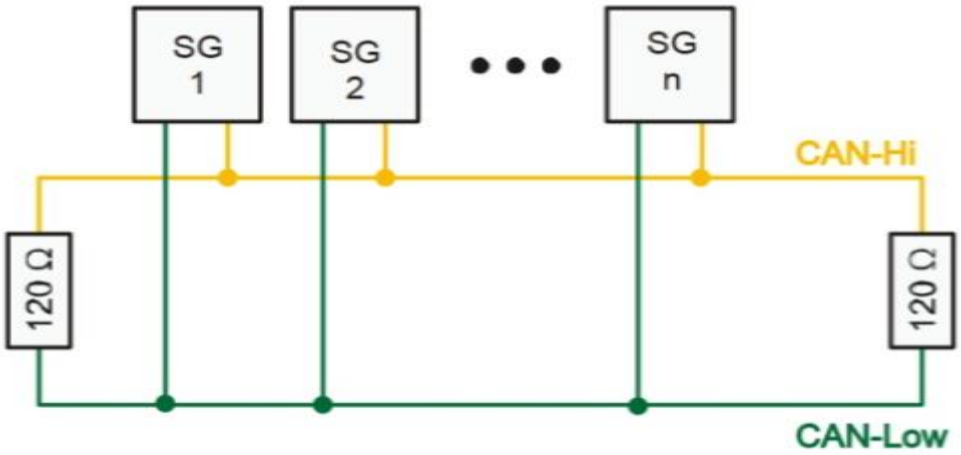

Fig 3. CAN connection. [16]

As shown in the graph, the CAN bus also differs from the LIN bus in that it relies on a two-wire physical implementation – Shielded Twisted Pair (STP) or Unshielded Twisted Pair (UTP) [19]. Because signal is represented by the voltage difference between the two wires, the two wires are closely twisted together to avoid voltage difference caused by disruption generated by external induction field [18] and the STP implementation also has the added protection from its mesh or foil shield [28]. Because of this robust signal representation method and the dynamic prioritization mechanism, CAN is much faster than LIN and more suitable for mission critical modules. CAN operates at minimum 1 MB/s while LIN operates at minimum 40 KB/s, as required by their specifications [19].

Message Frame

The CAN message frame consists of "fields", including "Start of Frame, Arbitration Field, Control Field, Data Field, CRC (Cyclical Redundancy Check) Field, ACK Field, and End of Frame" [16]. Inside the Arbitration Field, there is an identifier that is used to prioritize messages. When two nodes simultaneously send messages, the bits in their IDs will conflict, but such conflict is resolved by transmitting the dominant bit (logical 1) whenever there is a bit-wise conflict (both 0 and 1 are sent simultaneously, by different nodes). When a node notices a dominant bit at a time when it sends a recessive bit, this node will stop further transmission and yield to the higher priority ID [17]. This process is called bus arbitration and is illustrated by the graph below.

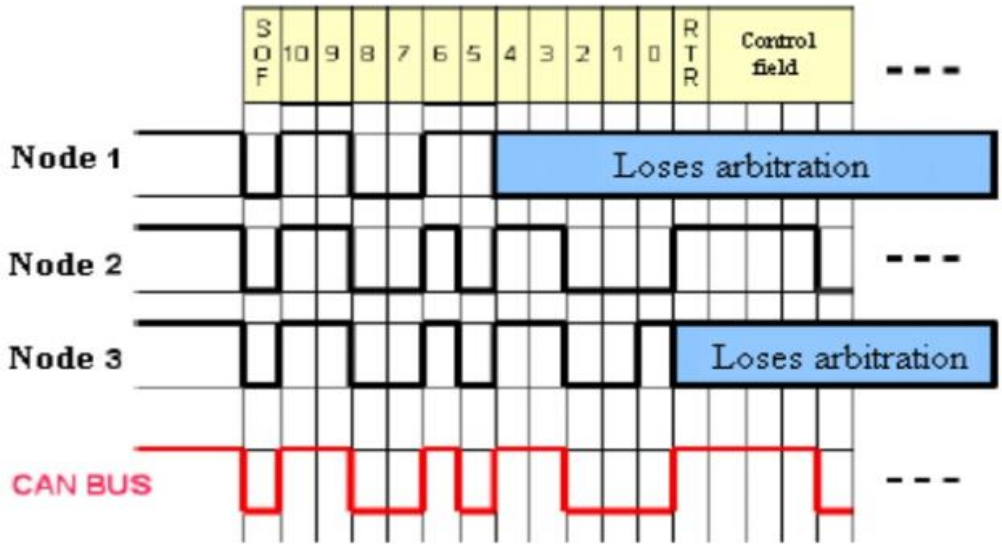

Fig 4. CAN bus arbitration [16]

Synchronization & Error Handling

Unlike LIN, CAN does not have a centralized time keeper, so every node has to synchronize their transmission for the arbitration to work. Besides internal time-keeping, CAN nodes re-synchronize themselves to the network at every transition from recessive to dominant bit [17].

Compared to LIN, CAN has many more error detection mechanisms. First, it has longer (16-bit, compared to 8-bit in LIN) checksum (cyclic redundancy code) calculated over many selected subfields of the preceding fields [19]. Second, it has an acknowledgement field filled by receiving node, which helps guarantee delivery. Third, if the receiver detects any error, including any message format errors, it can immediately send back an error frame [16]. All these active feedback mechanism make CAN's error detection much faster and more robust than LIN's. Therefore, it is extensively used in modules critical to vehicle safety, including motor control, electronic parking brake and vacuum leak detection [13].

FlexRay

Topology & Hardware

FlexRay can be implemented using a variety of different topologies, including line topology (i.e. bus topology) and star topology [29] (see graph below). The most popular implementation is using the star topology because FlexRay is usually implemented for its high fault tolerance and the star topology offers an extra fault tolerance enhancements: the active star node sitting at the center of the network can stop error propagation from any individual branch [22].

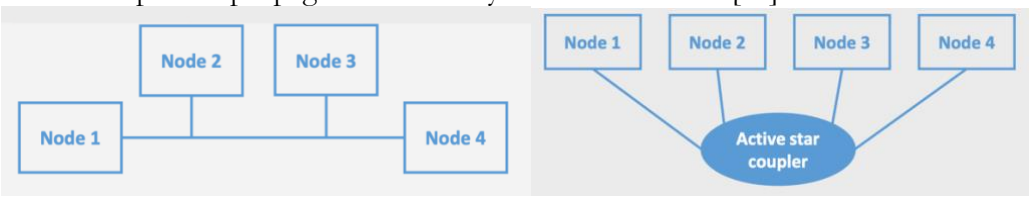

Fig 5. Line Topology vs. Star Topology. [29]

Like CAN, FlexRay's hardware is implemented on STP or UTP. However, each cable pair operates at 10MB/s bandwidth compared to 1MB/s in CAN [25]. More importantly, FlexRay is usually implemented with a redundant channel. If both channels are operational and used for unique data transmission, the total bandwidth is 20MB/s. The 10MB/s rate serves as a lower bound guarantee when one channel fails or if the network is configured to have redundancy [19]. Therefore, the actual topology is a derivative of the star topology (see graph below).

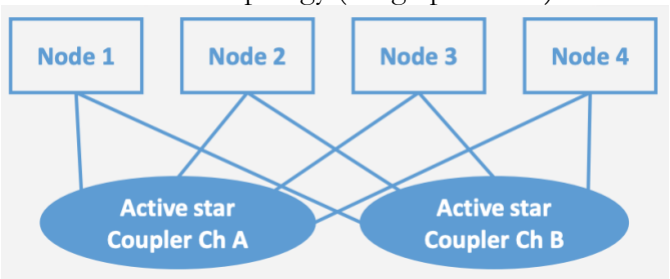

Fig 6. Redundant Star Topology. [29]

Message Frame

One of the biggest advantage of FlexRay over LIN and CAN, besides speed, is its dual use for both deterministic transmissions and event-triggered transmissions. The first graph below shows the a cycle consisting of static segment followed by a dynamic segment, with each slot (both static and dynamic) consisting of some idle time and a frame [33]. The second shows the detail of a frame.

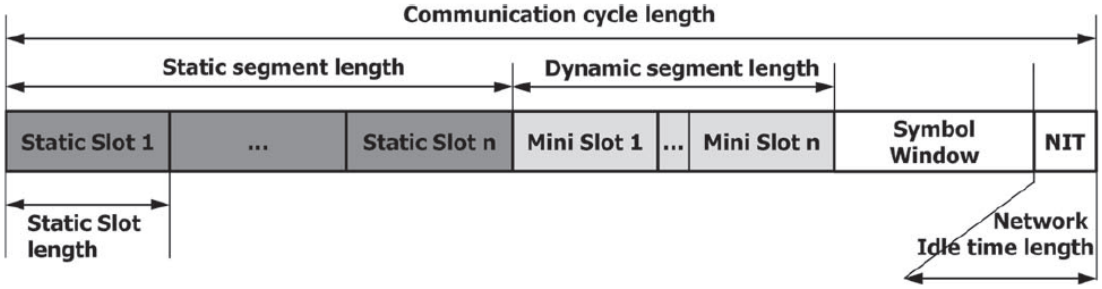

Fig 8. FlexRay communication cycle. [22]

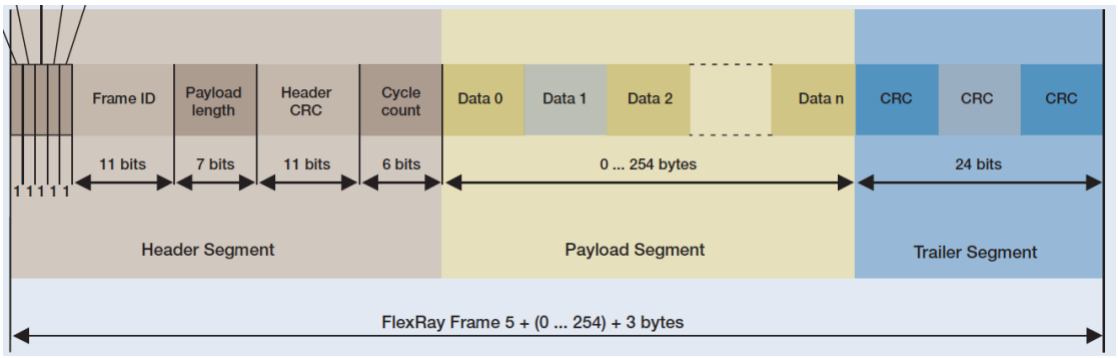

Fig 7. FlexRay message frame. [33]

Recall that the CAN network prioritizes messages dynamically based on ID, which opens the door for starvation of lower priority messages. Such delays may cause "critical accident" if such delays happen in the power train or chassis control [30]. FlexRay solves this issue by pre-allocating static slots for mission critical messages in the static segment of the frame, thus guaranteeing predictability in critical message propagation [22]. This strategy is called "time-triggered" communication as opposed to "event-triggered" communication [22].

FlexRay offers event-triggered communication through dynamic slots in addition to the time-triggered static slots. Compared to the highly predictable LIN network mediated by a single master, FlexRay offers more run-time flexibility to minimize idle time. Its dynamic segment is allocated using a rule similar to CAN's allocation scheme [25]. This dynamic segment is of variable size depending on how many nodes decide to utilize the segment time: All FlexRay nodes are aware of a pre-determined communication schedule that specifies the opportunity, but not necessity, to transmit. Then each node keeps a local counter to determine whose turn it is, if no send request is made by the corresponding node, each node increments the counter after one "mini-slot" of time. Otherwise, a "dynamic message" is sent. This process continues until either the counter passes all nodes' opportunities, or when the remaining time is not enough for a single dynamic message [31]. Note that there is an upper limit to the payload length.

Synchronization & Error Handling

Like CAN, FlexRay also has a global time shared by all nodes, and each node is responsible for synchronizing itself. The synchronization is done in a democratic fashion. Multiple nodes are set up to send a "sync message" in some pre-allocated static slots, and then an averages is calculated after excluding the outliers [32]. In between the synchronizations, FlexRay is also robust against small drifts within each message frame, because it uses a its sample window ("microtick" or cycle) is smaller than the bit transmission time [29, 25]. These two mechanisms give FlexRay a strong synchronization.

Like CAN, FlexRay message frames also carries a CRC (Cyclical Redundancy Check) code. It is calculated over all segments in the header segment and payload segment [19]. It is also 24-bit, which is even longer than the 16-bit in CAN. Though FlexRay lacks an explicit error reporting mechanism provided by error frames in CAN, some paper suggests the use of dynamic messages for error reporting to make up for it [30].

Ethernet

Topology & Hardware

While the updates from LIN to CAN, and then to FlexRay could be considered mostly quantitative improvements, the update from FlexRay to Ethernet really should be considered as a paradigm shift [26]. The previous updates increased the traffic speed by x100 by introducing better cables, balanced flexibility and predictability by innovating with time allocation schemes, and strengthened robustness by building more error check signals and hardware redundancies. However, all of LIN, CAN and FlexRay are message-based communication, which means that each message is broadcast to all receivers whether the message is relevant to them or not. The relevance is decided by nodes themselves after receiving the message. This implies that some participating nodes' time was wasted engaging in irrelevant communication, while they could have used the time to make meaningful exchanges among themselves. Ethernet solves this problem by placing a "switch" at the center of the star topology, selectively directing the messages to their relevant receivers only [35]. This effectively creates smaller "collision domains" by allowing parallel transmission of different messages at the same time [36].

Using a single UTP, the current vehicle Ethernet network is designed for 100MB/s bandwidth, which is x10 over FlexRay (when used in redundancy mode). Meanwhile, the IEEE802.3 task force is actively developing a new version of Ethernet network with gigabyte-level bandwidth [37, 38].

Message Frame

Because Ethernet messages can be selectively sent to certain nodes, each message frame contains a destination MAC address, indicating the physical address of the receiver node. The data payload length is dynamically decided as in FlexRay, but it can be much longer than the FlexRay payload (max 254 bytes), reaching 1500 bytes per frame [39]. This increase in payload size gives more flexibility to the payload content, allowing more straightforward serialization (and deserialization) by sender (and receiver) nodes because data compression becomes less important [27]. The SOME/IP (Scalable service-Oriented MiddlewarE over IP) protocol has been integrated into the AUTOSAR platform to work with the Ethernet network [41]. The SOME/IP protocol supports flexible serialization that can sometimes be as simple as an "one-to-one copy" into the payload [27].

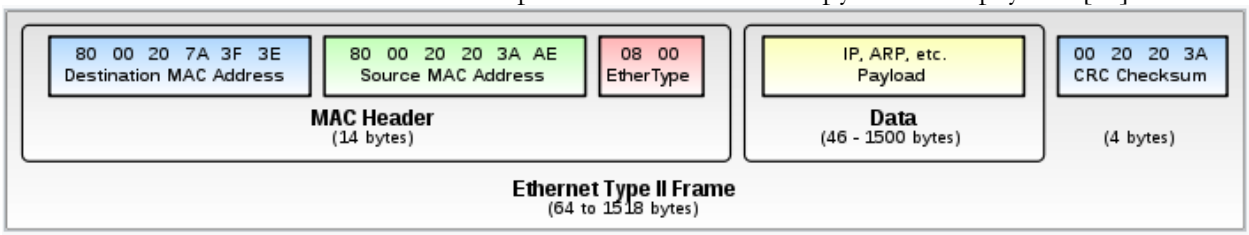

Fig 8. Ethernet message frame. [39]

It is very important to note that, unlike LIN, CAN, and FlexRay, the message frame of Ethernet is not being cyclically repeated constantly. Because the Ethernet messages are destination-based (rather than message-based like LIN, CAN, and FlexRay), the communication model can be service-based (rather than signal-based). The graph below illustrates the difference between service-based communication and signal-based communication. Note that while signal-based communication cyclically repeats the message frame, sending invalid values when no event occurs, service-based communication first publishes available service (channels for subscription) to all nodes, then accepts

subscriptions, and sends messages to subscribers only when an event occurs [27]. The series of operations are all supported by the SOME/IP protocol [41].

Fig 9. Signal-based communication vs. Service-based communication. [27]

Synchronization & Error Handling

Though Ethernet message transmission does not require close collaboration of multiple nodes at transmission time (i.e. different nodes filling in different segments of a message frame), synchronization is still required for certain activities over the network. For example, transmission of video and sound track for the infotainment system should be synchronized, and different sensors' data feeding into the fusion algorithms should be describing the same timestamp [27]. Ethernet uses a global time and the master node sends out periodic synchronization messages to all slave nodes. The synchronization is two-step, consisting of first a "sync message" and then the timestamp of the "sync message" both sent by the master [27].

Error Handling with Ethernet is a bit trickier. Though Ethernet has a longer CRC code (32-bit) [39] compared to FlexRay (24-bit), it also has a longer payload for the code to cover. To make it worse, Ethernet has a lower bound of 42-byte (46-byte when 802.1Q tag is absent [39]) for its payload, while the other networks allow a minimum of 8-byte payload [42]. Ethernet has this 42-byte lower bound because of its unique collision avoidance mechanism in an environment where many nodes can speak at once: the receiving node is responsible for notifying the conflicting senders to terminate transmission before this byte threshold. Therefore, any message with a smaller payload would be discarded as a collision [40].

A larger lower bound affects the reliability of the network negatively, because for mission critical messages, the user tend to use smaller payload to reduce "residual error probability" (the probability that a transmission error occurred despite correct CRC) [42]. One way to mitigate the poor "residual error probability" performance of Ethernet is adding an additional error check code into the payload data segment [42], which is a similar approach to how [30] solved the error reporting problem with FlexRay.

Conclusion

In this section, we examined four generations of in-vehicle bus networks that evolved in response to surging data volume. With each update, there had been innovative methods to increase bandwidth and reliability. Over the last few years, the introduction of Ethernet into the vehicle network system opened up vast potential for flexible, multi-gigabyte, service-based communication using adaptations (e.g. SOME/IP) of existing protocols in internet networks [37, 38]. However, as it has been pointed out in the "The Rise of Vehicle Bus" section, the newer generations of network tend to supplement, instead of fully replace, previous generations, due to high costs and the aforementioned open security problems [14, 27, 37]. Therefore, it is likely that we will continue to see LIN and CAN used in sub-parts of the data network in the near future, while the trend moves towards time-triggered predictable communication for mission critical messages using FlexRay [22], and service-based flexible communication for high-volume data using Ethernet [26, 27].

Aside from the insights into the future direction of vehicle bus system itself, the analysis also yields implications for both types of AV architecture. With respect to the software architecture, the increasingly heterogenous nature of the vehicle bus system further adds to the importance of a standardized runtime environment [22], so that lower-level communication can be abstracted from higher-level application developments, and can be easily interchanged as it evolves from one generation to the next. With respect to the functional architecture, as FlexRay and Ethernet bring higher bandwidth and runtime flexibility to inter-machine-communication, the hardware bottleneck is being increasingly shifted to the computing power of individual computing platform, favoring more distribution over more centralization [22].

Operating Systems

Overview

In the software architecture, the operating system sits next to the vehicle bus system in the "basic software layer" [22], at the bottom of the three-layer structure. Similar to the vehicle bus system, most AVs' operating system is heterogeneous in nature. In a distributed design, developers often selectively mount different types of OSes to the dozens of ECUs according to the ECUs' required real-time performance and runtime flexibility, as well as the responsible engineers' familiarity with the OSes [1, 22, 23]. In a centralized design, developers can even stack multiple OSes on top of each other to create different execution environments [34].

Types of OSes

Traditionally, as safety-critical systems, vehicles are mounted with OSes with strict real-time performance, pre-determined scheduling and static memory allocation [26]. These OSes are "deeply embedded" [26] into the minimal intelligence ECUs such as those for braking and steering. However, two newly emerging trends are changing the landscape of vehicle OS requirements.

First, with the rise of IoT movement, cars have expanded beyond their original transport functionality to become a part of people's connected life [72]. Increasingly, cars are being equipped with "infotainment" systems, providing services that have been previously provided by smartphones, such as "social media, web browsing, and even video chatting" [73]. These applications have vastly different requirements from the embedded ECUs for vehicle control, and usually build on general-purpose OSes such as Linux, Android, Apple OS and Windows [26, 34].

Second, as the car industry moves from lower levels of autonomous driving (level 0-3) to higher levels (level 4-5), the complexity and performance requirements of the embedded ECUs are drastically increasing [74]. Specifically, some of the new requirements include:
(1) integrate outputs from multiple sensor sources [74, 75]
(2) support a mix of processes with different levels of determinism, security and real-time requirements [77]
(3) make use of "acceleration platforms" [21] such as GPUs, FPGUs and ASICs [74, 76]
(4) offer automotive standard safety guarantee for "black box" algorithms used in AV cognitive modules [74]

Therefore, we can classify vehicle OSes into three categories. First, there are the traditional embedded real-time OSes (RTOS) [78] which can still be commonly found in the AV's control modules. Second, there are the general-purpose OSes for non-safety-critical infotainment modules, which offer great human interaction experience but do not need to be super reliable. Third, there are the RTOSes with enhanced capabilities, offering flexibility that could not be found in the traditional embedded RTOS [26].

GPOS for Infotainment

The technology for infotainment OS is not sharply innovative due to its non-critical nature. Despite the limited innovation, the infotainment OSes are not exact replicates of smartphone OSes. An important difference is that infotainment design needs to be less distractive for users before the arrival of fully autonomous driving [90]. On this account, Google' Android Auto and Apple's Carplay use their respective voice control Google Assistance and Siri to help [91]. However, according to an American Automobile Association (AAA) study, though Google and Apple's systems are demands less attention from users than the OEM proprietary systems, they still demand too much for safe use on road [92].

RTOS for AV of Level 4-5

QNX Neutrino, Wind River VxWorks and Green Hills Integrity, are the top three competitors in the space of RTOS for higher level AV, and they all have long tenures (20-30 years) for developing traditional embedded RTOS for cars. The solutions they provide share many similarities.

Firstly, they are all conform to a reduced set of POSIX API requirements [79, 80, 81], defined by the AUTOSAR Adaptive Platform [26]. This gives them a unique advantage over other RTOSes. Generally, RTOSes for embedded systems define their own set of proprietary APIs, making application migration and industry competition extremely hard [79]. POSIX systems, on the other hand, are widely standardized and highly competitive, but they are usually implemented as variants of UNIX, which is too bulky for embedded systems [79]. However, contrary to common belief, POSIX systems do not have to be UNIX-based. By definition, POSIX is simply a set of API specifications [82]. Therefore, QNX, Wind River, and Green Hills's products uniquely allow developers to port over their POSIX-based programs with PC origins to RTOSes that are appropriate for vehicle use.

Secondly, they are all built on a modular "microkernel" [83] architecture [79, 84, 85], which is very different from the "monolithic" UNIX architecture [82, 83]. "Microkernel" is a kind of lightweight OS design, where the OS only has the bare minimal functionality of managing address-space, process, thread, and interprocess communication [83, 86]. This minimalistic design makes the size of OS extremely small, with VxWorks 7.0 taking up merely 20 Kbyte [85]. The small size makes microkernel OSes suitable for some tiny-memory embedded computing units. On the other hand, they can also be easily scaled up to handle ECUs with high complexity [86]. The variety of functionalities offered by traditional OSes, such as filesystems, networking, device drivers, and GUI can be added onto the microkernel and managed as a regular process [83, 86]. This means that all the processes, including the "system processes", are now managed in user space [83]. This gives microkernel another advantage over monolithic OSes, which is "complete memory protection" for all processes, including the system processes that are traditionally run in kernel space [83].

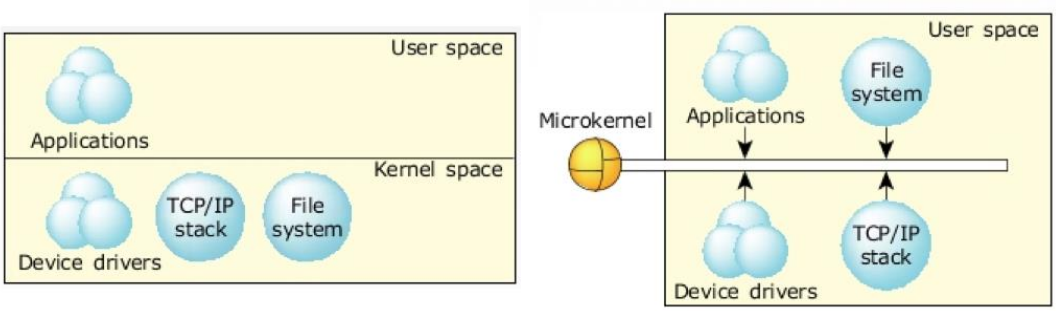

Fig 1. All processes run in user space in microkernel design (right), compared to monolithic design (left) [83]

Thirdly, they all offer real-time guarantees for critical processes through separation mechanisms. This feature had been automatic for the traditional RTOS with the sole purpose of running real-time tasks. Now it is important to manage real-time tasks side-by-side with dynamic non-critical tasks on the same OS, while still upholding the guarantee. QNX claims that its microkernel design gives it "the inherent ability to separate multiple domains spatially and temporally at the application level", and therefore enabling the dual use for both real-time and non-real-time applications on a single OS [77]. VxWorks protects critical processes by allocating "a predetermined number of CPU cycles… and space partition" "in addition to preemption", but it also allows more flexible scheduling options for less critical processes [88]. Similar to VxWorks, Green Hills Integrity also reserve CPU cycles for critical processes in its scheduling scheme [84]. In addition, Green Hills protects the system from "memory exhaustion" with its unique "memory quota system". All system calls issued by processes do not use the kernel memory. Instead, "messages, semaphores, or other kernel objects" live in the defined memory space of the process that issued the call, so that no malicious process can overrun the kernel memory [80].

Finally, all three OSes are certified by the ISO 26262 standard [36], which is a functional safety standard for all road vehicle software, and the proof of compliance to which can be very arduous. All three vendors offer the ISO 26262 ASIL D certificate to be purchased alongside the OS software [74, 77, 88]. It is especially difficult to certify the safety of "black box" algorithms such as neural network [74]. According to Green Hills, it can be done by running a "plausibility analysis function" with provable safety alongside the black box and resolving any conflict using a "decision function" [74].

Industry Insights

Development Trajectories

From the surveyed literature, it is clear that the industry as well as academia are both only at the very beginning of a journey towards standardized functional architecture for AV systems. The lack of standardization is hampering competitive development of interchangeable modules, which in turn entrenches divergent proprietary designs. However, as the competition intensifies after AVs are commercialized to mass market, we can hope for a dominant design to emerge and trigger standardization. As for now, the best we could do is to open up the discussion for best practices in AV architecture designs, such as safety separation, de-centralization, data latency management, upgrade/downgrade compatibility, and redundant dataflows. Regardless of the particular proprietary design, these design features as demonstrated by their implementations should benefit most stakeholders by making AV systems safer, faster, and more scalable.

On the other hand, the software architecture in AV has a much clearer three-layer design, shared across different platforms. Therefore, the foundational infrastructure of the software architecture – the vehicle bus system and operating system have much clearer development trends, with the vehicle bus system moving into a hybrid design relying on Ethernet and service-oriented communication as its powerhouse, and with the operating system clustering around three specialized uses of deeply embedded ECUs, infotainment, and high-performance ECUs for complex cognition. These trends are largely driven by the move towards higher levels of autonomy in the industry, and its resulting increase in demand for flexibility. As the software architecture has come much further on the evolution curve, a next step would be to devise and enforce more appropriate safety standards on the software platform. The simple lack of rigorous standards on latency [21], and the awkward inapplicability of traditional standards on safety [74] could prove costly as companies continues to evolve their designs without the opportunity to build in the standards from ground up.

A Case Study of Infotainment OS

The infotainment system represents the higher end of speed and flexibility requirement of automotive OS, as its functionality differs drastically from the traditional embedded real-time OS for low level control modules. At the same time, the technology readiness is higher for infotainment system than the many other modules needed for future autonomous vehicles, such as perception and planning, because the infotainment fulfills functions similar to smartphones, and its performance is not critical to the safety of the driving system. Therefore, the infotainment OS is currently one of the most mature market for new generation vehicle software, and serve as a good case study for possible trajectories of other vehicle software markets.

The competitive space features three types of players: The first type is OS developer from the consumer technology world, such as Apple, Google, and Microsoft. The second type is OEM developing proprietary systems for their own brand of cars, such as Toyota and Hyundai [89]. The third type is specialized automotive software supplier such as QNX.

Though OEMs and specialized automotive software suppliers entered the market earlier, they are increasingly being pushed out by late-comers from the consumer technology world [89]. One key differentiation for infotainment OSes is their associated application ecosystem. Unlike in other ECUs, where automotive-grade OSes are abstracted away from functional module developers,

infotainment OSes selectively approve applications developed by OS-specific third-party developers [90]. The infotainment OS companies can therefore use the different approval standard to build unique user experiences as a key aspect of competition [93]. The higher quality of app developers on the platforms of consumer technology companies is their other major advantage, helping them gain leverage over OEMs, turning OEMs from competitors with proprietary infotainment OSes into business partners and members of their own consortiums [94].

Standard Alliances

The future of autonomy is close, but an industry-wide effort towards standardization is required to make it safe. Alliances such as the AUTOSAR consortium have been effective in standardizing the software architecture and in establishing a common language for OEMs' suppliers to communicate design differences. With the timely release of AUTOSAR Adaptive, such alliance has also proved to be an effective model for facilitating technological paradigm shifts (i.e. ethernet, POSIX-based OS) throughout the industry.

At the same time, as shown by the case study of infotainment OS, as the requirements for automotive modules shifts from solely reliability and security to focus more on intelligence and agility, new outsider forces such as consumer software companies, AI/data companies and robotics companies can also pop up into the competition, bringing unique skillsets and forging their own module-specific consortiums. In the case of infotainment OS, this means a waste of development time and resource spent by OEMs and automotive-specific suppliers in trying to make something that is not a part of their core competence. Therefore, any universally adopted AV standard should be less of a tool to shield away competition, but more of an architectural solution for modifiability and interoperability that helps the industry expects new entrant forces.

References

[1] Behere, Sagar, and Martin Törngren. "A Functional Reference Architecture for Autonomous Driving." *Information and Software Technology* 73 (2016): 136–50. https://doi.org/10.1016/j.infsof.2015.12.008.

[2] Pendleton, Scott, Hans Andersen, Xinxin Du, Xiaotong Shen, Malika Meghjani, You Eng, Daniela Rus, and Marcelo Ang. "Perception, Planning, Control, and Coordination for Autonomous Vehicles." *Machines* 5, no. 1 (2017): 6. https://doi.org/10.3390/machines5010006.

[3] "Lidar 101: An Introduction to Lidar Technology, Data, and Applications." coast.noaa.gov. National Oceanic and Atmospheric Administration (NOAA) Coastal Services Center, 2012. https://coast.noaa.gov/data/digitalcoast/pdf/lidar-101.pdf.

[4] Kim, Junsung, Hyoseung Kim, Karthik Lakshmanan, and Ragunathan (Raj) Rajkumar. "Parallel Scheduling for Cyber-Physical Systems." *Proceedings of the ACM/IEEE 4th International Conference on Cyber-Physical Systems - ICCPS '13*, 2013. https://doi.org/10.1145/2502524.2502530.

[5] Salgian, G., and D.h. Ballard. "Visual Routines for Autonomous Driving." *Sixth International Conference on Computer Vision (IEEE Cat. No.98CH36271)*, 1998. https://doi.org/10.1109/iccv.1998.710820.

[6] Mcnaughton, Matthew, Christopher R. Baker, Tugrul Galatali, Bryan Salesky, Christopher Urmson, and Jason Ziglar. "Software Infrastructure for an Autonomous Ground Vehicle." *Journal of Aerospace Computing, Information, and Communication* 5, no. 12 (2008): 491–505. https://doi.org/10.2514/1.39487.

[7] Bass, Len, Paul Clements, and Rick Kazman. *Software Architecture in Practice*. Upper Saddle River, NJ: Addison-Wesley, 2015.

[8] Gowdy, Jay. *A Qualitative Comparison of Interprocess Communications Toolkits for Robotics*. Pittsburgh, PA: Carnegie Mellon University, the Robotics Institute, 2000.

[9] Innovations, Real-Time. "DDS in Autonomous Car Design - Whitepaper." RTI. Accessed March 2, 2020. https://www.rti.com/dds-in-autonomous-car-design_wp.

[10] AUTOSAR Consortium. "AUTOSAR Layered Software Architecture." www.autosar.org, 2008. http://www.autosar.org/download/specs_aktuell/-AUTOSAR_LayeredSoftwareArchitecture.pdf.

[11] "Vehicle Bus." Wikipedia. Wikimedia Foundation, March 1, 2020. https://en.wikipedia.org/wiki/Vehicle_bus.

[12] DiagnoseDan. "Lin-Bus Free Educational Video," August 1, 2017. https://www.youtube.com/watch?v=tkaD_jomweA.

[13] Staff, Embedded. "Re-Evaluating the Role of the LIN Bus in Vehicle Sensor and Control Applications." Embedded.com, November 2, 2014. https://www.embedded.com/re-evaluating-the-role-of-the-lin-bus-in-vehicle-sensor-and-control-applications/.

[14] K S, Sanal, and Kanpur Rani. "Hierarchical Automobile Communication Network Using LIN Subnet with VI Based Instrument Clusters and OBD-II Regulations." *International Journal of Scientific & Technology Research* 1, no. 3 (April 2012): 36–39.

[15] "Local Interconnect Network." Wikipedia. Wikimedia Foundation, March 1, 2020. https://en.wikipedia.org/wiki/Local_Interconnect_Network#LIN_protocol.

[16] Tarek, Radwa. "Automotive Bus Technologies." LinkedIn SlideShare, October 24, 2016. https://www.slideshare.net/RadwaTarek7/automotive-bus-technologies.

[17] "CAN Bus." Wikipedia. Wikimedia Foundation, March 1, 2020. https://en.wikipedia.org/wiki/CAN_bus.
[18] DiagnoseDan. "Can-Bus Trouble," May 13, 2018. https://www.youtube.com/watch?v=uKnQI2IScPU.
[19] "Debugging CAN, LIN and FlexRay Automotive Buses with an Oscilloscope." Tektronix, February 4, 2020. https://www.tek.com/document/application-note/debugging-can-lin-and-flexray-automotive-buses-oscilloscope.
[20] Liu, Shaoshan, Liyun Li, Jie Tang, Shuang Wu, and Jean-Luc Gaudiot. *Creating Autonomous Vehicle Systems*. San Rafael, CA: Morgan & Claypool Publ., 2018.
[21] Lin, Shih-Chieh, Yunqi Zhang, Chang-Hong Hsu, Matt Skach, Md E. Haque, Lingjia Tang, and Jason Mars. "The Architectural Implications of Autonomous Driving." *ACM SIGPLAN Notices* 53, no. 2 (2018): 751–66. https://doi.org/10.1145/3296957.3173191.
[22] Jo, Kichun, Junsoo Kim, Dongchul Kim, Chulhoon Jang, and Myoungho Sunwoo. "Development of Autonomous Car—Part I: Distributed System Architecture and Development Process." *IEEE Transactions on Industrial Electronics* 61, no. 12 (2014): 7131–40. https://doi.org/10.1109/tie.2014.2321342.
[23] Jo, Kichun, Junsoo Kim, Dongchul Kim, Chulhoon Jang, and Myoungho Sunwoo. "Development of Autonomous Car - Part II: A Case Study on the Implementation of an Autonomous Driving System Based on Distributed Architecture." *IEEE Transactions on Industrial Electronics* 62, no. 8 (2015): 5119–32. https://doi.org/10.1109/tie.2015.2410258.
[24] Goebl, Matthias, and Georg Farber. "A Real-Time-Capable Hard-and Software Architecture for Joint Image and Knowledge Processing in Cognitive Automobiles." *2007 IEEE Intelligent Vehicles Symposium*, 2007. https://doi.org/10.1109/ivs.2007.4290204.
[25] "FlexRay." Wikipedia. Wikimedia Foundation, February 26, 2020. https://en.wikipedia.org/wiki/FlexRay#cite_note-2.
[26] Furst, Simon, and Markus Bechter. "AUTOSAR for Connected and Autonomous Vehicles: The AUTOSAR Adaptive Platform." *2016 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshop (DSN-W)*, 2016. https://doi.org/10.1109/dsn-w.2016.24.
[27] Ziehensack, Michael. "Webinar: Ethernet the New Generation of ECU Communication - HD." Elektrobit, February 3, 2015. https://www.youtube.com/watch?v=aWwVRq7HW64.
[28] Goyal, Shubham, Joel, and R Eswar. "Difference Between UTP and STP Cables (with Comparison Chart)." Tech Differences, December 28, 2019. https://techdifferences.com/difference-between-utp-and-stp-cables.html.
[29] Murvay, Pal-Stefan. "FlexRay." Universitatea Politehnica Timisoara, November 18, 2018. http://www.aut.upt.ro/~pal-stefan.murvay/teaching/nes/Lecture_08_FlexRay.pdf.
[30] Yoon, Seung-Hyun, Suk-Hyun Seo, Jin-Ho Kim, Key Ho Kwon, Sung-Ho Hwang, and Jae Wook Jeon. "A Method to Handle CRC Errors on the Basis of FlexRay." *2009 IEEE International Symposium on Industrial Electronics*, 2009. https://doi.org/10.1109/isie.2009.5213765.
[31] "FlexRay_E: Dynamic Segment." Learning. Accessed March 16, 2020. https://elearning.vector.com/mod/page/view.php?id=397.
[32] "FlexRay_E: Synchronization Method." Learning. Accessed March 16, 2020. https://elearning.vector.com/mod/page/view.php?id=405.
[33] EDN. "Evaluation Kit Can Kick Start FlexRay Applications -." EDN, October 23, 2005. https://www.edn.com/evaluation-kit-can-kick-start-flexray-applications/.

[34] Haydin, Victor. "Everything You Wanted to Know About Types of Operating Systems in Autonomous Vehicles." Intellias, October 15, 2019. https://www.intellias.com/everything-you-wanted-to-know-about-types-of-operating-systems-in-autonomous-vehicles/.
[35] Cisco & Cisco Router. "What Are Ethernet and Ethernet Switches? - Cisco & Cisco Network Hardware News and Technology." Cisco & Cisco Network Hardware News and Technology. Cisco & Cisco Network Hardware News and Technology, February 24, 2015. http://ciscorouterswitch.over-blog.com/article-what-are-ethernet-and-ethernet-switches-99288225.html.
[36] "Network Switch." Wikipedia. Wikimedia Foundation, March 17, 2020. https://en.wikipedia.org/wiki/Network_switch#cite_note-6.
[37] Levi, Ziv. "Automotive Ethernet: The Future of In-Car Networking?" StackPath, April 4, 2018. https://www.electronicdesign.com/markets/automotive/article/21806349/automotive-ethernet-the-future-of-incar-networking.
[38] "Automotive Ethernet: An Overview." https://www.ixiacom.com/. ixia, May 2014. https://support.ixiacom.com/sites/default/files/resources/whitepaper/ixia-automotive-ethernet-primer-whitepaper_1.pdf.
[39] "Ethernet Frame." Wikipedia. Wikimedia Foundation, March 13, 2020. https://en.wikipedia.org/wiki/Ethernet_frame.
[40] Robb, Ben. "Why Is There a Minimum Ethernet Framesize?" A few selections from the archives, January 10, 2007. https://benrobb.com/2007/01/10/why-is-there-a-minimum-ethernet-framesize/.
[41] Voelker, Lars. "Scalable Service-Oriented MiddlewarE over IP (SOME/IP)." Scalable service-Oriented MiddlewarE over IP (SOME/IP). Accessed March 18, 2020. http://some-ip.com/.
[42] Rahmani, Mehrnoush, Wolfgang Hintermaier, Bernd Muller-Rathgeber, and Eckehard Steinbach. "Error Detection Capabilities of Automotive Network Technologies and Ethernet - A Comparative Study." *2007 IEEE Intelligent Vehicles Symposium*, August 13, 2007. https://doi.org/10.1109/ivs.2007.4290194.
[43] Tas, Omer Sahin, Florian Kuhnt, J. Marius Zollner, and Christoph Stiller. "Functional System Architectures towards Fully Automated Driving." *2016 IEEE Intelligent Vehicles Symposium (IV)*, 2016. https://doi.org/10.1109/ivs.2016.7535402.
[44] Zong, Wenhao, Changzhu Zhang, Zhuping Wang, Jin Zhu, and Qijun Chen. "Architecture Design and Implementation of an Autonomous Vehicle." *IEEE Access* 6 (2018): 21956–70. https://doi.org/10.1109/access.2018.2828260.
[45] Serban, Alexandru Constantin, Erik Poll, and Joost Visser. "A Standard Driven Software Architecture for Fully Autonomous Vehicles." *2018 IEEE International Conference on Software Architecture Companion (ICSA-C)*, 2018. https://doi.org/10.1109/icsa-c.2018.00040.
[46] Liu, Shaoshan, Jie Tang, Zhe Zhang, and Jean-Luc Gaudiot. "Computer Architectures for Autonomous Driving." *Computer* 50, no. 8 (2017): 18–25. https://doi.org/10.1109/mc.2017.3001256.
[47] Kato, Shinpei, Shota Tokunaga, Yuya Maruyama, Seiya Maeda, Manato Hirabayashi, Yuki Kitsukawa, Abraham Monrroy, Tomohito Ando, Yusuke Fujii, and Takuya Azumi. "Autoware on Board: Enabling Autonomous Vehicles with Embedded Systems." *2018 ACM/IEEE 9th International Conference on Cyber-Physical Systems (ICCPS)*, 2018. https://doi.org/10.1109/iccps.2018.00035.
[48] International Organization for Standardization (ISO). "ISO Standard 26262:2011 Road Vehicles - Functional Safety," 2011.

[49] Leigh, Bob, and Reiner Duwe. "Software Architecture of Autonomous Vehicles." *ATZelectronics Worldwide* 14, no. 9 (2019): 48–51. https://doi.org/10.1007/s38314-019-0104-7.
[50] Bellairs, Richard. "What Is ISO 26262? An Overview." Perforce Software, January 3, 2019. https://www.perforce.com/blog/qac/what-iso-26262-overview.
[51] Koopman, Philip, Uma Ferrell, Frank Fratrik, and Michael Wagner. "A Safety Standard Approach for Fully Autonomous Vehicles." *Lecture Notes in Computer Science Computer Safety, Reliability, and Security*, 2019, 326–32. https://doi.org/10.1007/978-3-030-26250-1_26.
[52] Rauhamäki, Jari, Timo Vepsäläinen, and Seppo Kuikka. "Functional Safety System Patterns." *Proceedings of VikingPloP 2012 Conference* 22 (March 20, 2012).
[53] Bonnefon, J.-F., A. Shariff, and I. Rahwan. "The Social Dilemma of Autonomous Vehicles." *Science* 352, no. 6293 (2016): 1573–76. https://doi.org/10.1126/science.aaf2654.
[54] VanWashenova, Jeff. "To Centralize or Distribute - That Is The Question." FierceElectronics, October 19, 2018. https://www.fierceelectronics.com/components/to-centralize-or-distribute-question.
[55] Clarke, Daniel. "Centralized or Decentralized? Autonomous Vehicles Are Forcing Key Architectural Decisions." Design News, December 28, 2018. https://www.designnews.com/electronics-test/centralized-or-decentralized-autonomous-vehicles-are-forcing-key-architectural-decisions/28764088459995.
[56] Mcauslin, Allan. "Safe Autonomous Systems: Centralized or Distributed Processing?" NXP Blog, September 27, 2018. https://blog.nxp.com/automotive/safe-autonomous-systems-centralized-or-distributed-processing.
[57] "Advanced Driver-Assistance System Trends and Challenges." Mitchell, June 23, 2019. https://www.mitchell.com/mitchellnews/detail/articleid/3597/Advanced-Driver-Assistance-System-Trends-and-Challenges-.
[58] Shankland, Stephen. "Take a Close-up Look at Tesla's Self-Driving Car Computer and Its Two AI Brains." CNET. CNET, August 20, 2019. https://www.cnet.com/news/meet-tesla-self-driving-car-computer-and-its-two-ai-brains/.
[59] Kunz, Felix, Dominik Nuss, Jurgen Wiest, Hendrik Deusch, Stephan Reuter, Franz Gritschneder, Alexander Scheel, et al. "Autonomous Driving at Ulm University: A Modular, Robust, and Sensor-Independent Fusion Approach." *2015 IEEE Intelligent Vehicles Symposium (IV)*, 2015. https://doi.org/10.1109/ivs.2015.7225761.
[60] Korosec, Kirsten. "Waymo to Start Selling Standalone LiDAR Sensors." TechCrunch. TechCrunch, March 6, 2019. https://techcrunch.com/2019/03/06/waymo-to-start-selling-standalone-lidar-sensors/.
[61] Nuss, Dominik, Manuel Stuebler, and Klaus Dietmayer. "Consistent Environmental Modeling by Use of Occupancy Grid Maps, Digital Road Maps, and Multi-Object Tracking." *2014 IEEE Intelligent Vehicles Symposium Proceedings*, 2014. https://doi.org/10.1109/ivs.2014.6856516.
[62] Wei, Junqing, Jarrod M. Snider, Tianyu Gu, John M. Dolan, and Bakhtiar Litkouhi. "A Behavioral Planning Framework for Autonomous Driving." *2014 IEEE Intelligent Vehicles Symposium Proceedings*, 2014. https://doi.org/10.1109/ivs.2014.6856582.
[63] Maurer, M., and E.d. Dickmanns. "A System Architecture for Autonomous Visual Road Vehicle Guidance." *Proceedings of Conference on Intelligent Transportation Systems*, 1998, 578–83. https://doi.org/10.1109/itsc.1997.660538.
[64] Ademovic, Adnan. "An Introduction to Robot Operating System: The Ultimate Robot Application Framework." Toptal Engineering Blog. Toptal, March 3, 2016. https://www.toptal.com/robotics/introduction-to-robot-operating-system.

[65] Wille, Marcel, and Ulrich Kleine. "Volkswagen Goes Adaptive." its-mobility. Volkswagen AG, 2017. https://www.its-mobility.de/download/FuSi/Dokumentation/Wille_Folien_FuSi_2017.pdf.
[66] "Welcome to Apache ZooKeeper™." Apache ZooKeeper. Accessed April 22, 2020. https://zookeeper.apache.org/.
[67] Admin. "Difference between a CPU and a Core." CPU vs Core - Difference Between. TheyDiffer.com, February 20, 2018. https://theydiffer.com/difference-between-a-cpu-and-a-core/.
[68] "CPU vs GPU: Definition and FAQs." OmniSci. Accessed April 23, 2020. https://www.omnisci.com/technical-glossary/cpu-vs-gpu.
[69] "Digital Signal Processor." Wikipedia. Wikimedia Foundation, April 14, 2020. https://en.wikipedia.org/wiki/Digital_signal_processor.
[70] "Application-Specific Integrated Circuit." Wikipedia. Wikimedia Foundation, March 10, 2020. https://en.wikipedia.org/wiki/Application-specific_integrated_circuit.
[71] "Field-Programmable Gate Array." Wikipedia. Wikimedia Foundation, April 7, 2020. https://en.wikipedia.org/wiki/Field-programmable_gate_array.
[72] Miller, Jessica. "Wind River Delivers Solutions for Next Generation ADAS and Autonomous Vehicle Innovation." Wind River. Accessed April 28, 2020. https://www.windriver.com/news/press/pr.html?ID=13772.
[73] "In-Car Entertainment." Wikipedia. Wikimedia Foundation, February 27, 2020. https://en.wikipedia.org/wiki/In-car_entertainment.
[74] "Green Hills Platform for Safe and Secure Automated Driving Systems." Green Hills Software. Accessed April 28, 2020. https://www.ghs.com/products/auto_adas.html.
[75] "QNX News Releases." QNX UNVEILS NEW SOFTWARE PLATFORM FOR ADAS AND AUTOMATED DRIVING. Accessed April 28, 2020. http://www.qnx.com/news/pr_6298_1.html.
[76] "QNX Platform for ADAS." BlackBerry QNX. Accessed April 28, 2020. https://blackberry.qnx.com/en/software-solutions/automotive/qxn-adas.
[77] "QNX OS For Safety Automotive - ProductBrief," 2019. https://blackberry.qnx.com/content/dam/qnx/products/certified_os/QNX-OS-For-Safety-Auto-Print_ProductBrief_2019.pdf.
[78] "Real-Time Operating System." Wikipedia. Wikimedia Foundation, April 16, 2020. https://en.wikipedia.org/wiki/Real-time_operating_system.
[79] "The Philosophy of QNX Neutrino." Help - QNX CAR 2 Documentation. Accessed April 29, 2020. http://www.qnx.com/developers/docs/qnxcar2/index.jsp?topic=%2Fcom.qnx.doc.neutrino.sys_arch%2Ftopic%2Fintro.html.
[80] "Real-Time Operating System." Green Hills Software. Accessed April 29, 2020. https://www.ghs.com/products/rtos/integrity.html#reliability.
[81] "VxWorks." Wind River. Accessed April 29, 2020. https://www.windriver.com/products/vxworks/.
[82] "An Embeddable POSIX OS?" Help - QNX CAR 2 Documentation. Accessed April 29, 2020. http://www.qnx.com/developers/docs/qnxcar2/index.jsp?topic=%2Fcom.qnx.doc.neutrino.sys_arch%2Ftopic%2Fintro_embeddable.html.
[83] "Microkernel." Wikipedia. Wikimedia Foundation, April 3, 2020. https://en.wikipedia.org/wiki/Microkernel.

[84] "Integrity The Most Advanced RTOS Technology." Accessed April 28, 2020. https://www.cs.unc.edu/~anderson/teach/comp790/papers/INTEGRITY_RTOS.pdf.
[85] Cole, Bernard. "Wind River Brings a 20 Kbyte Microkernel to the VxWorks RTOS." Embedded.com, March 3, 2014. https://www.embedded.com/wind-river-brings-a-20-kbyte-microkernel-to-the-vxworks-rtos/.
[86] "Product Scaling." Help - QNX CAR 2 Documentation. Accessed April 29, 2020. http://www.qnx.com/developers/docs/qnxcar2/topic/com.qnx.doc.neutrino.sys_arch/topic/intro_Product_scaling.html.
[87] "Microkernel Architecture." Help - QNX CAR 2 Documentation. Accessed April 29, 2020. http://www.qnx.com/developers/docs/qnxcar2/topic/com.qnx.doc.neutrino.sys_arch/topic/intro_MICROKERNELARCH.html.
[88] "VxWorks Product Overview." Wind River. Accessed April 29, 2020. https://resources.windriver.com/vxworks/vxworks-product-overview.
[89] Jang, Seung-ju. "Development Trend of Operating System Technology for Smart Car." *International Journal of Recent Trends in Engineering & Research* 3, no. 01 (2017).
[90] Stevens, Tim. "2014's Battle for Dashboard Supremacy: Apple's CarPlay vs. Google's OAA vs. MirrorLink." Roadshow, March 4, 2014. https://www.cnet.com/roadshow/news/2014s-battle-for-dashboard-supremacy-apples-carplay-vs-googles-oaa-vs-mirrorlink/.
[91] Teague, Chris. "Android Auto Vs. Apple CarPlay." Digital Trends. Digital Trends, November 27, 2019. https://www.digitaltrends.com/cars/android-auto-vs-apple-carplay/.
[92] Esplin, Jess, M. Loveless, Kelly L. Mackenzie, Connor J. Motzkus, and Camille L. Wheatley. "Visual and Cognitive Demands of Using Apple CarPlay, Google's Android Auto and Five Different OEM Infotainment Systems." AAA Foundation, February 3, 2020. https://aaafoundation.org/visual-cognitive-demands-apples-carplay-googles-android-auto-oem-infotainment-systems/.
[93] Favell, Andy. "Everything You Need to Know about Building Apps for Connected Cars." ClickZ, November 28, 2019. https://www.clickz.com/everything-you-need-to-know-about-building-apps-for-connected-cars/95810/.
[94] Paukert, Chris. "Toyota Finally Rolls out Android Auto at Chicago Auto Show." Roadshow. CNET, February 7, 2019. https://www.cnet.com/roadshow/news/toyota-android-auto-chicago-auto-show/.
[95] Bloor, Thomas, and Bob Leigh. "Path to Building Autonomous Car Architectures: RTI Webinars." Path to Building Autonomous Car Architectures | RTI Webinars, 2020. https://www.rti.com/low-risk-path-to-building-autonomous-car-architectures.
[96] California Department of Motor Vehicles. "Key Autonomous Vehicle Definitions," 2020. https://www.dmv.ca.gov/portal/dmv/detail/vr/autonomous/definitions.
[97] "The 5 Levels of Autonomous Vehicles." TrueCar Blog, January 24, 2020. https://www.truecar.com/blog/5-levels-autonomous-vehicles/.
[98] "Advanced Driver-Assistance Systems." Wikipedia. Wikimedia Foundation, February 11, 2020. https://en.wikipedia.org/wiki/Advanced_driver-assistance_systems.
[99] "By 2030, 25% of Miles Driven in US Could Be in Shared Self-Driving Electric Cars." https://www.bcg.com, April 10, 2017. https://www.bcg.com/d/press/10april2017-future-autonomous-electric-vehicles-151076.