TECHNISCHE UNIVERSITÄT KAISERSLAUTERN

ELEKTROTECHNIK UND INFORMATIONSTECHNIK

Remote Control von Peripheriekomponenten in einem Fahrzeugnetz

Kamel Fakih

Masterarbeit 2025

Rheinland-Pfälzische Technische Universität Kaiserslautern Landau

Politecnico di Torino

Masterarbeit | Master's Thesis

Remote Control von Peripheriekomponenten in einem Fahrzeugnetz

Remote Control of Peripheral Components in an Automotive in-vehicle Network

Author: Kamel Fakih

Matriculation number: 428403

Submission date: 4. März 2025

Chair of Electronic Design Automation

Examiner: Prof. Dr.-Ing. Wolfgang Kunz

Prof. Dr. Matteo Sonza Reorda

Supervisor: Dr. Naresh Nayak

Adriaan Neiß

Eidesstattliche Erklärung

Ich versichere hiermit an Eides statt, die vorliegende Arbeit gemäß BPO/MPO Elektrotechnik und Informationstechnik bis auf die durch meinen Betreuer gewährte Unterstützung ohne Hilfe Dritter selbstständig angefertigt, alle benutzten Quellen und Hilfsmittel einschließlich des Internets vollständig und genau angegeben und alles kenntlich gemacht zu haben, was aus Arbeiten anderer unverändert, mit Abkürzung oder sinngemäß übernommen wurde.

Kaiserslautern, den 4. März 2025

Kamel Fakih

Kurzfassung

Die Automobilindustrie bewegt sich in Richtung softwaredefinierter Fahrzeuge (SDV) – Ein neues Paradigma, bei dem die Kernfunktionen und Merkmale eines Fahrzeugs durch Software definiert werden. Dieser Übergang geht einher mit dem Übergang zu zentralisierten elektrischen und elektronischen Architekturen (EEA), wie der zonale Architektur. Eine Schlüsseleigenschaft der Zentralisierung ist die Konsolidierung von Rechenressourcen im Fahrzeugcomputer, die darauf abzielt, das Systemdesign zu vereinfachen und gleichzeitig fortschrittliche Funktionen wie Over-the-Air-Updates zu ermöglichen. Die Zentralisierung ermöglicht die Vereinfachung der Fahrzeug-Peripheriegeräten, indem Softwarekomponenten, die auf speziellen Sensor-/Aktor-Steuergeräten (ECUs) laufen, in die Fahrzeugcomputer und/oder die zonale Steuergeräte verlagert werden. Infolgedessen werden diese Steuergeräte durch vereinfachte Edge Nodes ersetzt, auf denen minimale Software läuft und die als Gateways zwischen der Anwendungslogik und den Sensoren/Aktuatoren fungieren. Die Edge Nodes führen Low-Level-Befehle, wie SPI-Transaktionen aus, die sie vom Fahrzeugcomputer/zonalen Controller erhalten. Ein Remote Control-Protocol (RCP) wird zur Interaktion mit diesen Knoten über Ethernet verwendet. Das Protokoll definiert einheitliche Methoden für den Zugriff auf und die Steuerung von Peripheriegeräten, die an Edge Nodes angeschlossen sind. RCP befindet sich noch in der Standardisierungsphase und ist nicht vollständig definiert. Daher wurden in dieser Arbeit Remote Control Use-Cases innerhalb der zonalen Architektur untersucht, um ein Framework für RCP zu erstellen, das Systemmodell zu definieren und die Anforderungen für das Kommunikationsprotokoll abzuleiten.

Diese Forschung wurde bei der Robert Bosch GmbH unter der Aufsicht von Dr. Naresh Nayak und Adriaan Neiß von Bosch, Professor Wolfgang Kunz von der RPTU und Professor Matteo Sonza Reorda von Polito durchgeführt.

Abstract

The automotive industry is shifting towards software-defined vehicles – A new paradigm in which software defines the core functions and features of a vehicle. This transition is accompanied by the move towards centralized electrical and electronic architectures (EEA), such as the zonal architecture. A key property of centralization is the consolidation of compute resources in the vehicle computer, which aims to simplify the system design while enabling advanced features such as over-the-air updates. Centralization simplifies the vehicle peripherals by moving software from dedicated sensor/actuator electronic control units (ECUs) to the vehicle computers and/ or zonal controllers. As a result, these ECUs could be replaced by simplified edge nodes that act as gateways between the application logic and the sensors/actuators. The edge nodes execute low-level commands, like SPI transactions, received from the vehicle computer/zonal controller. A remote control protocol (RCP) interacts with these nodes over Ethernet. The protocol defines uniform methods for accessing and controlling the peripherals connected to the edge nodes. RCP is still under standardization and is not fully defined. Therefore, in this thesis, we explored remote control use cases within the zonal architecture to establish a framework for RCP, defining the system model and deriving the requirements for its communication protocol. Then, we evaluated two candidate transport protocols – IEEE1722 and SOME/IP – and how they would fit into the RCP framework. We learned that while IEEE1722 is more suitable as a transport mechanism for RCP because it is lightweight and more compatible with time-sensitive networking, it failed short of satisfying the service-oriented nature of some RCP requirements as interface discovery. Therefore, we defined a new concept that combines components from each protocol to realize a feature-complete remote control protocol.

This research was conducted at Robert Bosch GmbH under the supervision of Dr. Naresh Nayak and Mr. Adriaan Neiß from Bosch, Professor Wolfgang Kunz from RPTU, and Professor Matteo Sonza Reorda from Polito.

Inhaltsverzeichnis

1 Introduction

Trends in automotive industry

The automotive industry is undergoing a paradigm shift in which vehicles are no longer considered a basic means of transportation but rather a fully adaptable experience that caters to the different needs of drivers and passengers. When looking for a new vehicle, today's customers are less influenced by performance metrics, such as horsepower and torque. Instead, they are more interested in the availability of digital functions like driver assistance, infotainment, and connectivity [1]. Studies have shown that 36% of customers are willing to switch to a different car manufacturer if it offers improved digital and connected services [2].

With digital devices becoming an integral part of the end users' lives, they now demand seamless integration of their vehicles into their digital lifestyle. Users expect regular updates to improve the driver and passenger experience, along with the ability to add or remove features on demand. In a fully connected world, vehicles are expected to automatically sync with schedules, routes, and preferences, like adjusting the driver's seating position or playing their favorite music upon entry [3]. In other words, assuming the availability of the required hardware, any vehicle should adapt itself to a given driver profile, similar to how smartphones adjust to individual user settings. The shift towards designing driver experience-oriented vehicles will make software a key player in the automotive industry. While hardware components are still necessary for vehicles to operate, they are no longer sufficient for providing the needed flexible and improved user experience [4]. In traditional vehicles, software components were only the enabler for mechanical functions. Software was deployed on statically configured hardware to implement basic functionality, and it remains unchanged throughout the vehicle's lifetime. In contrast, software in future vehicles will become the sufficient condition required to define the vehicle's main functionalities. The software will dynamically change a vehicle's functionality while accommodating new releases and changes in hardware components. The global software market is estimated to have a faster growth rate than the whole automotive market as it doubles in value from 2020 to 2030 [2].

This trend has led to the rise of a new term: the SDV (software defined vehicle), which is, by definition, a vehicle that has its main features and functionalities driven by software [1]. SDVs represent a ground-breaking approach to transportation, utilizing software integration and virtualization to improve vehicle functionality, connectivity, and autonomy [5]. The popularity of

SDVs is driven by factors like over the air updates, driver assistance systems, and connectivity:

1. **OTA (over the air) updates**: The ability to perform OTA updates is one of the key enablers for SDVs. It allows OEMs to deliver software updates and patches and add new features to a vehicle remotely without the need to roll it back into the shop. Performing these updates has several advantages, such as efficiency and cost savings. Manufacturers greatly reduce downtime and operational costs by remotely updating a large fleet of vehicles. Regular bug fixes and patches for newly discovered security vulnerabilities also help protect the vehicle from evolving threats, enhancing the overall security of the vehicle and its passengers. An SDV can continue to adapt to new technologies and users' needs long after it leaves the factory. [5]

2. **ADAS (advanced driver assistance system):** Architectural changes brought by SDVs led to the fast-paced evolution of technologies assisting the driver. Features such as adaptive cruise control, lane assist, automatic braking, and blind-spot detection all rely on complex software algorithms and need to process huge amounts of data coming from a variety of sensors [5]. These features underscore the role of software in ensuring a safe and smooth driving experience.

3. **Connectivity:** Modern vehicles rely on connectivity to integrate seamlessly into their ecosystem. V2V communication (vehicle-to-vehicle communication) improves road safety by enabling cooperative functionalities like platooning and exchanging critical safety data. V2I communication (vehicle-to-infrastructure communication) improves efficiency and convenience by allowing vehicles to communicate with smart infrastructure, improving route planning. Finally, V2C communication (vehicle-to-cloud communication) gives the vehicle access to navigation data, real-time traffic information, and personalized services [5]. Connectivity makes SDVs a part of an adaptive, efficient, and intelligent ecosystem.

4. **Economic and Competitive Edge** Due to the differences in regional regulatory frameworks, it is difficult to harmonize and develop uniform technical regulations for vehicles; for instance, the US automotive safety standards are regulated by the NHTSA, which focuses more on compliance with federal regulations, while the European safety standards are set by the European Commission, and are based on government regulatory approval before manufacturing [6]. This often leads to different vehicle designs and adaptations

to meet the respective local/regional regulations (e.g., the US regulations require vehicles to be locked in "Park"before key removal to prevent roll-away, while other countries permit the removal of the keys while in neutral to make it possible to roll the vehicle while keyed-off in automated parking garages [7]). OEMs aims to boost profitability by cutting costs and shortening the time to market. The high number of variants in vehicles is still a major cost driver for development that is promised to be reduced by prioritizing software at the core of vehicle design [3]. Additionally, SDVs can bring extra revenue streams after the initial purchase by offering connectivity features using a subscription-based model, given that such features offer adequate value and improve the driving experience [8].

Changes Enabled by software defined vehicles

SDVs represents a shift from traditional vehicles' tightly integrated software and hardware architecture to a layered and decoupled architecture. Software in a traditional vehicle's component subsystem (e.g., engine control system) is directly embedded within its hardware components, leading to a solidified and more robust vehicle implementation. However, to promote flexibility and modularity, SDVs adopts a layered architecture on which the cross-dependency between different technical elements is reduced. Figure 1.1 summarizes the technical components of an SDV. The key components include functional hardware, the EEA (electrical and electronic architecture), computing platform, OS (operating system), communication middleware, service layer, functional and service applications, and the cloud service platform. The onboard hardware encompasses functional hardware, including sensors for data generation and actuators for carrying out commands. The electrical and electronic architecture defines the IVN (in-vehicle network) that interconnects all functional hardware to the computing platform, which is responsible for most data processing inside the vehicle. The onboard software consists of the OS kernel, the communication middleware, and the service layer, with the kernel managing the components' low-level hardware, while the middleware facilitates communication across the system by providing a unified data aggregation platform. The service layer abstracts the discrete and repeated vehicle functions as services that can be accessed by the upper software layers, namely the functional and service applications. Functional applications involve hardware control and security, while service applications usually take care of infotainment and other services, such as charging. Finally, the cloud service platform is responsible for sending some of the data generated inside the vehicle to the cloud for further analysis to improve the user experience [4].

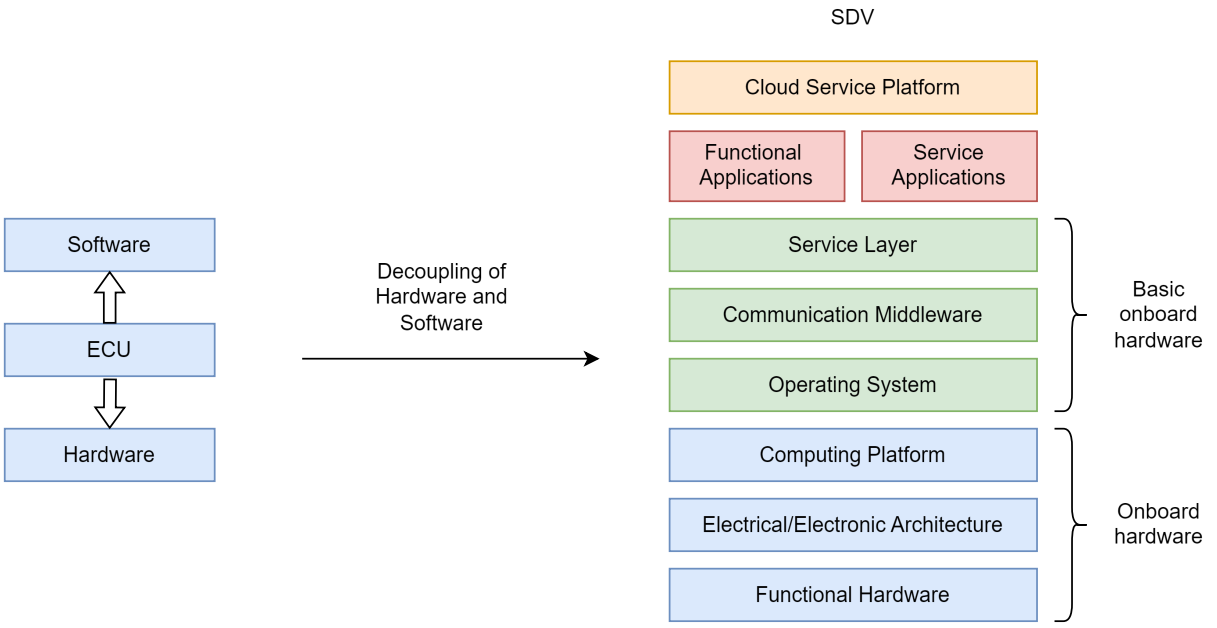


Fig. 1.1: Technical elements of an SDV (adapted from [4])

The primary value of traditional vehicles originated from their hardware components, as the software was only embedded in hardware to complement its functionality. However, for SDVs, hardware becomes only a shared resource with its main functionality defined by software [4]. To implement the complex features required by today's vehicles, system components require access to various sensors and actuators, some of which may belong to other subsystems. Thus, it is necessary to move from the current paradigm where component subsystems are often treated as black boxes and implement open and standardized interfaces for these components instead. Abstracting the functional hardware facilitates continuous software updates and allows the hardware to be upgraded or exchanged in the future.

Another change introduced by SDVs is the adoption of new in-vehicle electrical and electronic architectures. Traditional decentralized EEAs are increasingly struggling to meet the demands of the complex functional requirements of modern vehicles. In a decentralized EEA, advanced vehicle functions are achieved by a high number of interconnected ECUs (electronic control units) such that each ECU is responsible for processing its own set of information and communicating with other ECUs using point-to-point communication. Such a design adds too much complexity and suffers from many pitfalls regarding flexibility and scalability. It cannot handle the high data exchange bandwidth required by advanced features like ADAS. For this reason, modern vehicles are shifting towards a centralized EEA, a novel system design that con-

solidates the computation of functions at various levels, such as individual domains or zones within the vehicle [7]. Figure 1.2 highlights the shift from the traditional decentralized EEA to a zonal EEA. While we are going to explore the progression of electrical and electronic architectures in the automotive industry in more detail, in short, a zonal EEA, consolidates ECUs by grouping functions and connections into zones based on the vehicle's physical layout. For instance, the vehicle in Figure 1.2 is grouped into four zones: front left, front right, rear left, and rear right zone. In contrast to the decentralized network on the left, the zonal controllers on the right communicate with the central compute platform using automotive Ethernet. Communication in traditional EEAs is based on legacy communication protocols, such as CAN (controller area network), LIN (local interconnect network), FlexRay, and MOST (media oriented system transport); however, these protocols are becoming inadequate against the increased data volume required by modern vehicles. On the other hand, automotive Ethernet is getting increasingly adopted due to its ability to handle high volumes of data and its development to support time-sensitive communication [7].

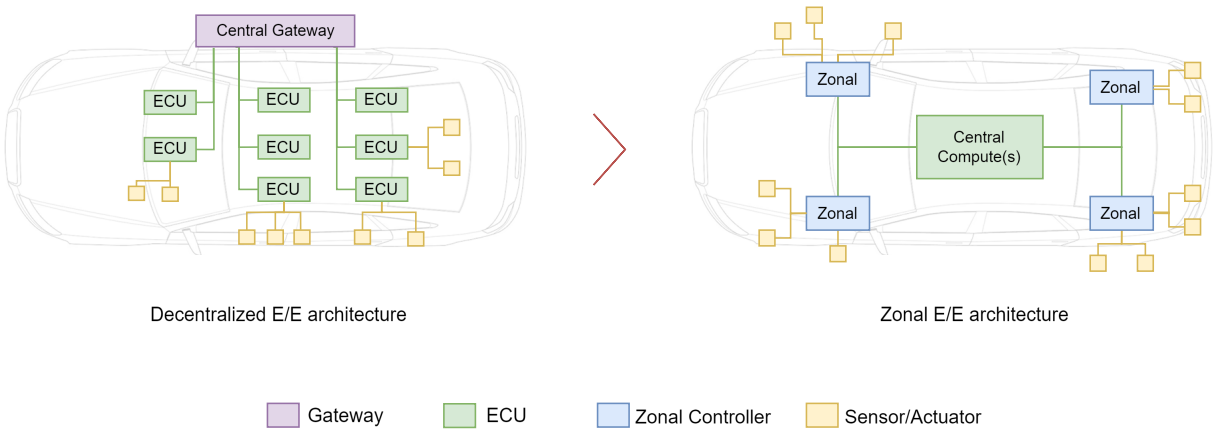


Fig. 1.2: Transition to a centralized electrical and electronic architecture

The centralization of software presents a significant opportunity to simplify system design. By concentrating software in the central compute platform, the complexity of edge nodes, which include the sensors and actuators distributed through the vehicle, is significantly reduced. Edge nodes are no longer required to have extensive processing power or complete software stacks, as their primary role is to collect and send raw data to the central ECU. Reducing complexity also creates the opportunity for seamless OTA updates. To perform an OTA update, the OEM releases a new service pack containing the latest software version. The service pack is then downloaded wirelessly over the network by the vehicle's CCU (connectivity control

unit) and passed on to the central computing unit, which hosts the onboard update server that is responsible for performing the firmware updates on the various ECUs across the in-vehicle network [9]. The simplification of the edge nodes further facilitates the process, as the nodes require minimal to no adjustments during updates. Reducing the number of ECUs makes the entire vehicle feature deployment process more efficient and less prone to errors. This also offers promising cost savings and enables potential reductions in wiring harness length.

However, relying on low-cost edge nodes with minimal software creates the need for a new and reimagined approach to interaction between the functional hardware and the compute platform within the vehicle. As sensors and actuators become simpler and less powerful, the central computing platform takes on the responsibility of managing their operation, creating the need to implement a new approach for controlling vehicle's peripherals. The central ECU is then required to transparently manage interfaces like SPI, I2C, and GPIO over Ethernet. This setup can be described as a remote control system containing a set of nodes, where one node, the controller, is capable of accessing the peripherals connected to other nodes, the edge nodes while relying on Ethernet as their main communication channel. Considering the variety of interfaces and their different configurations, it is logical to define a standard that describes system operation, including initialization, configuration, diagnostics, communication, and error handling. This standard can be implemented through a remote control protocol that defines a uniform method for accessing and controlling the remote interfaces. The protocol should govern the data formats and interactions between nodes in the remote control system. Eventually, the controller must be able to perform various operations on the remote interfaces, such as reading their current state, getting/setting configuration parameters, and executing their core functionality.

Figure 1.3 shows an example use case of an RCP (remote control protocol). In this setup, the SPI interface on the edge node is connected to 3 different SPI sensors. The application logic for handling and processing sensor data is relocated to the central ECU. The application expects to seamlessly access the SPI interface as if directly connected to the central ECU. Therefore, the edge node must act as a gateway, forwarding commands from the central ECU over the Ethernet interface to the SPI interface. RCP standardization is still a work in progress, meaning that in this scenario, the exact behavior and communication between the edge node and the central ECU is still undefined. However, a set of requirements for RCP can be outlined by considering use cases from OEMs and potential remote control system architectures. RCP definition is met with a variety of challenges, including identifying the interfaces that need to be

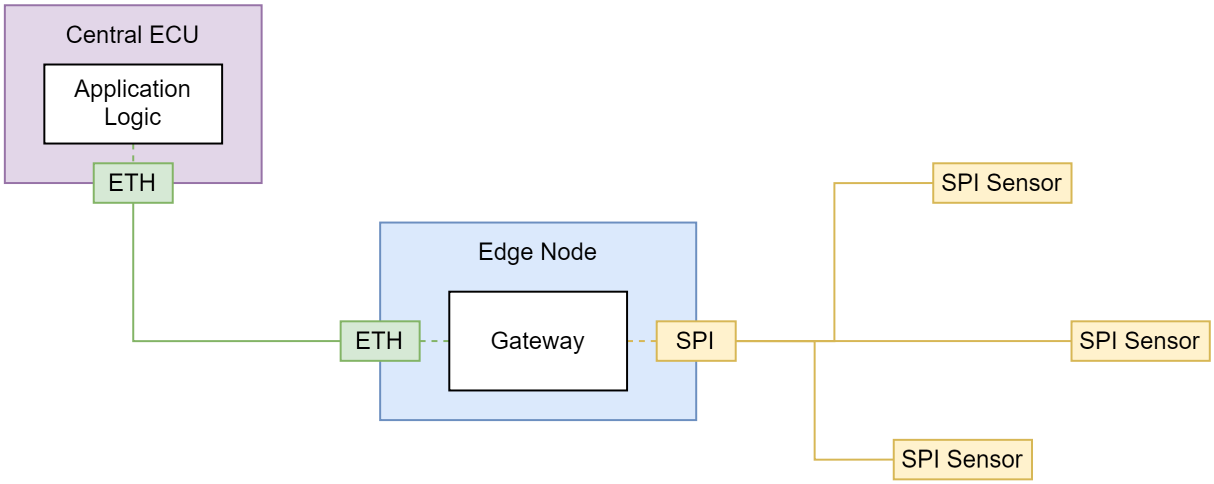


Fig. 1.3: Example of an RCP application

remotely controlled, specifying the type of operations, and addressing communication properties. Additionally, a remote control system must meet performance requirements, including low latency and quick startup times, along with stringent security and functional requirements. As a communication protocol, RCP is expected to define a set of messages that govern the system behavior. For example, the protocol might include a data polling message that the controller uses to perform a read operation on the remote interface. Additionally, the protocol will need an underlying transport mechanism that facilitates the delivery of these messages between the different participants of the system.

A major decision when defining RCP is whether to come up with a completely new transport mechanism or to rely on existing communication protocols that meet these requirements. Several candidates for an underlying transport mechanism already exist, including IEEE1722, which is a steaming protocol designed for low-latency communication over Ethernet, and SOME/IP, which is a communication middleware that supports message-based communication and service discovery over Ethernet/IP networks. This thesis aims to evaluate these protocols based on a feature-to-cost analysis. The thesis aims to identify the suitability of existing communication protocols as underlying transport mechanisms for RCP by examining their unique features and applying them to different RCP use cases. Furthermore, the thesis will identify the main features required for an efficient and functional remote control system, exploring the possibility of combining existing protocols to create a complete RCP solution. Finally, the thesis will provide recommendations to support the ongoing standardization efforts for RCP

The thesis is organized as follows: Chapter 2 presents the state of the art of automotive

networks, protocols, and RCP. Chapter 3 presents a framework for remote control protocols, defining the system model, use cases, and requirements. Chapter 4 discussed the implementation of existing in-vehicle communication protocols as RCP. Chapter 5 presents a concept for a new remote control protocol based on our evaluation results, and finally, Chapter 6 concludes the thesis.

2 State of the art

2.1 electrical and electronic architectures

To address the ambiguity around the definition of the vehicle's electrical and electronic architectures, the paper in [9] relies on IEEE's general definition of an architecture to describe it as:

> "the fundamental organization of vehicle electrical and electronic components, including ECUs, sensors, actuators, wiring, power distribution, onboard and wireless communication, etc., to realize the desired function and performance goals, with emphasis on the interactions and interdependencies among the components and with the environment, as well as the principles guiding the design and evolution."

This means that the EEAs serves as the foundational blueprint for developing the vehicle's E/E components. A well-designed architecture allows the vehicle to achieve its performance goals and adapt to future requirements while enabling seamless refinement and expansion of its functions. It also affects the overall system design process, which includes requirements, design considerations, objectives, and development strategies [9]. The rapid increase in modern vehicle's functional requirements has led to a surge in the number of components that need to interact within the vehicle. For instance, modern vehicles may include over a hundred ECUs and up to 100 million lines of codes [10]. This increase in complexity has introduced significant challenges in designing the in-vehicle network. To fully understand these challenges, it is imperative to examine the evolution of the electrical and electronic architectures over time and to explore their future developement path.

2.1.1 Traditional decentralized electrical and electronic architectures

Vehicles till the 1960s relied on minimal electrical components with no electronics. Electrical connections were mainly required for basic functions like operating the headlights and sound systems. It was not until the 1970s that vehicles started to include more electronic components. During this phase, the positioning of the wiring harness – which includes the wires, connectors, and terminals that transmit power and data signals within the vehicle - started to become a concern for the system designers. Driven by new requirements for safety, emissions, and energy savings from regulators, electronic component growth maintained momentum through the

1980s and 1990s. This growth was coupled with an increase in the vehicle's data signals, which reached bandwidths of up to 25MB/s. As a result, it became more important to define standards for electrical architectures that would help optimize the vehicle's performance and reduce the design complexity [11].

These two decades witnessed the introduction of an extensive amount of electronic control systems, such as electronic fuel injection, ignition and emission systems, anti-lock braking, airbags, anti-collision systems, GPS navigation, and fault diagnosis systems. Many of these systems used dedicated ECUs, with each ECU handling a specific application(e.g., electronic power steering). Figure 2.1 shows an example of such architecture. The communication between ECUs

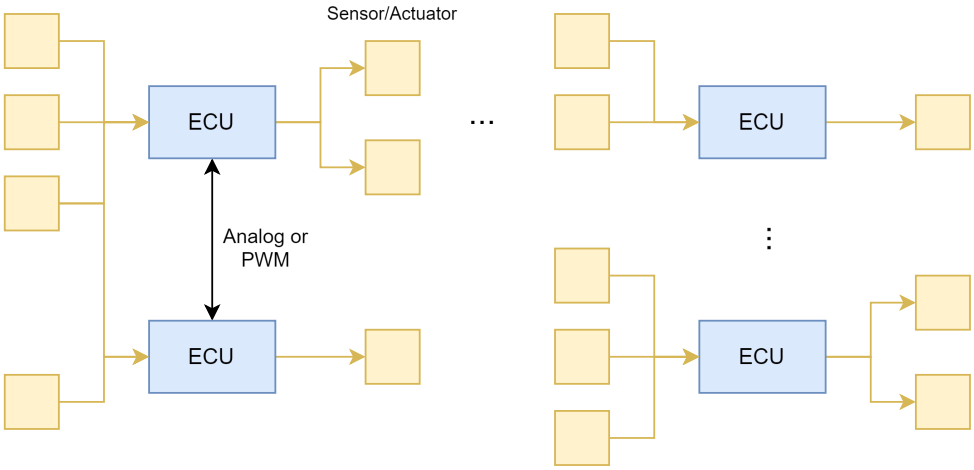


Fig. 2.1: Example of a point-to-point EEA

was limited to a few use cases, on which ECUs relied on point-to-point connections like analog and PWM signals to exchange information about sensors shared across multiple components. However, with the demand for more shared information and diagnostics data, the wiring harness started to become unmaintainable and less reliable. In response, manufacturers opted for field bus technologies as an alternative to point-to-point communication, reducing the number of communication lines inside the vehicle. The communication between multiple ECUs would be carried over a limited number of buses capable of interconnecting more than two devices and can ensure their proper coordination. This has led to the introduction of multiple fieldbus technologies, including the CAN-bus, LIN, MOST, and FlexRay, which served different uses within the in-vehicle network. Vehicle functions were then categorized into various domains, such as powertrain, chassis, body, and infotainment. Eventually, ECUs and sensors of a common domain were connected to a serial bus, forming a subnetwork as shown in Figure 2.2. Inter-domain communication was still limited but possible as ECUs could act as gateways when needed. This

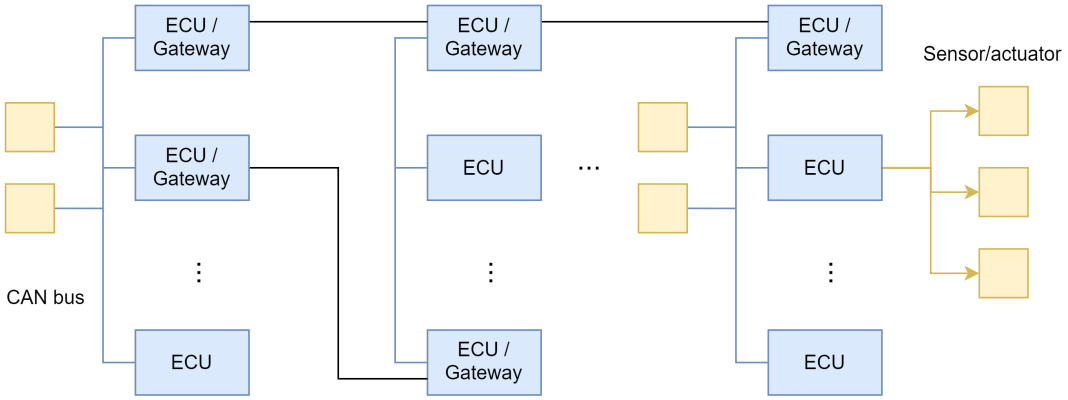


Fig. 2.2: Example of a distributed EEA integrated with vehicle bus

approach facilitated function design and implementation by confining the developement of closely related functions to their domain. However, as more coordination was required from different functional domains, the system later evolved to rely on a central gateway that handles all the inter and intra-domain communication. Figure 2.3 illustrates the main concept of this type of EEA. The central gateway allowed communication between different bus types and was respon-

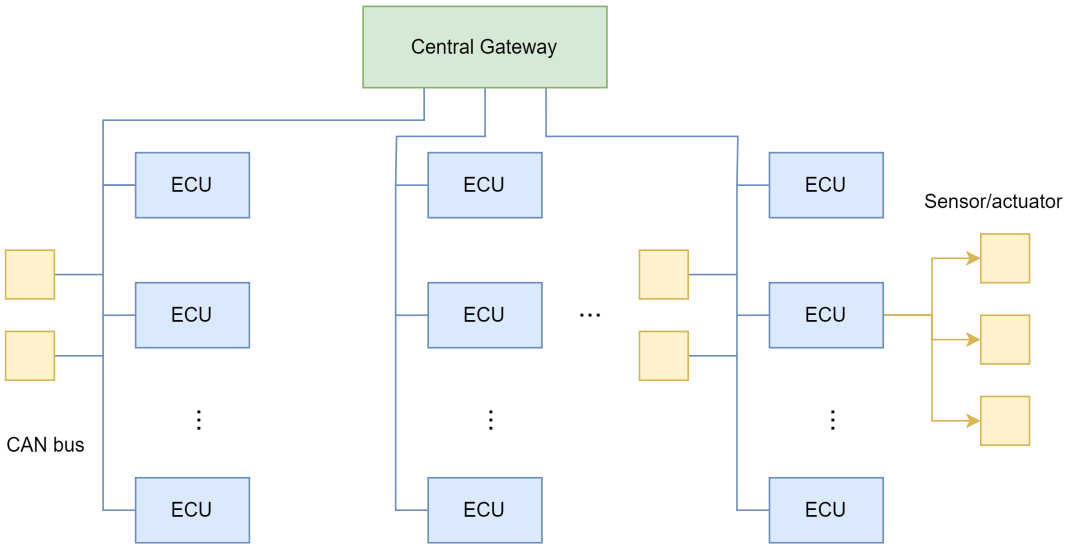


Fig. 2.3: Example of an EEA with centralized gateway

sible for additional tasks such as message routing, rate adaptation, network management, and diagnostics. This architecture was implemented by major OEMs including Volkswagen, BMW, and Audi, and remained in use until recently[9, 12].

Up until this point, vehicle control was still fully decentralized. Although inter-ECU communication became a thing, generally, ECUs was limited to mainly one function per ECU. This approach presented some advantages—for example, decentralized architectures allowed for the

seperation of concerns. By deploying tightly related functions into a single ECU, it became easier to integrate that ECU into the network as the bandwidth and timing requirements on the bus became simpler (i.e. eliminates the need for synchronization between multiple ECUs). However, this approach also faced several drawbacks. Adding more ECU for each function soon proved unmaintainable and expensive. As the number of functions within the vehicle increased, so did the number of ECUs, which led to bulkier wiring harnesses. Additionally, bus technologies started to reach their upper limit, as the volume of data between ECUs started to increase [7].

2.1.2 Centralized electrical and electronic architectures

The traditional decentralized approach for EEA could no longer support the growth in content and complexity of modern vehicles. New technical trends in automated driving and connectivity significantly increased the demand for communication within and across different functional domains, placing more load on the central gateway. Excessive decentralization of vehicle functions resulted in complex wiring and poor network performance. As a result, the vehicles EEAs required a fundamental change, which led to the introduction of new architectures focusing on centralization. Initially, domain-oriented architectures were introduced to reduce the number of ECUs and improve the gateway load. Followed by the development of zone-oriented architectures to further consolidate vehicle functions [12].

Domain-oriented architectures

Figure 2.4 presents a typical domain architecture layout. In a domain-oriented architecture, software components are grouped based on their functional domain. For example, the system could be organized into the following domains:

- **Body and Cabin:** Includes user-related functionality such as lighting and climate control.
- **Infotainment:** Includes cabin displays, entertainment, and information systems such as navigation and connectivity features.
- **Vehicle Motion and Safety:** Includes chassis safety and driver assistance functions.
- **Powertrain:** Includes drive systems, transmission, and energy management.

Each domain is composed of a domain control unit (DCU) and one or more subdomain ECUs connected to it. The DCU is a powerful computing unit that hosts domain-specific functionalities and serves as an abstraction layer for the underlying ECUs, facilitating inter-domain com-

munication. On the other hand, subdomain ECU abstracts the functionalities of the sensors and actuators they are connected to. The subdomain ECU, along with their corresponding sensor and actuators, are usually referred to as smart sensors and actuators [7].

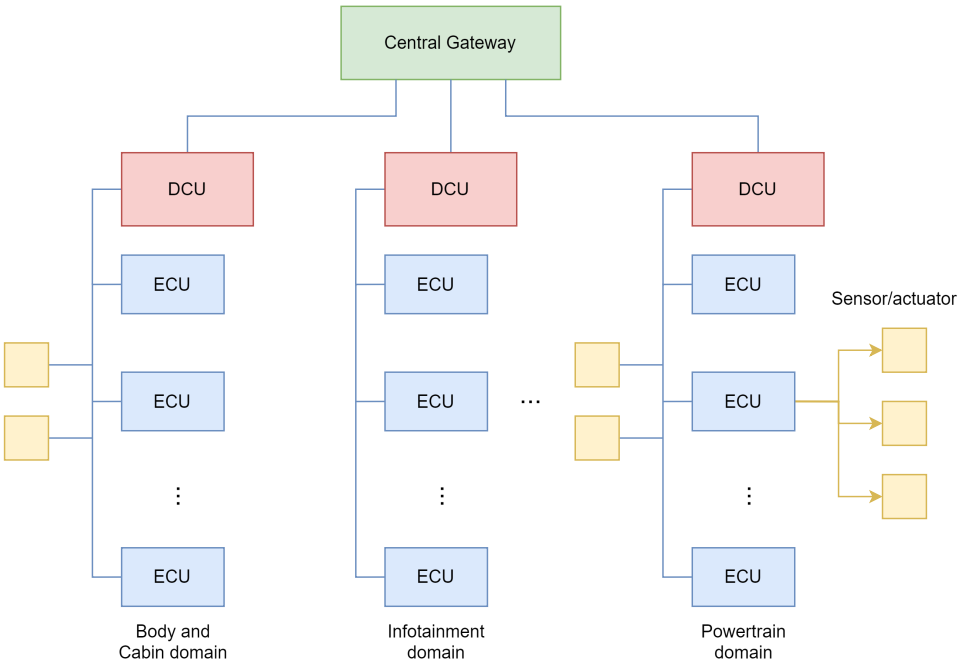


Fig. 2.4: Typical domain-oriented EEA

Zone-oriented architectures

Figure 2.5 presents the layout of a zonal architecture. Zone-oriented architectures divide the car into zones based on their proximity. For example, a zonal architecture could be split into front right, front left, rear right, and rear left zones. Similarly, a Zone consists of a Zone ECU (ZCU) and on or more ECUs. However, A key difference between Zonal and Domain-oriented architectures is that the sub-zone ECUs performs less processing than sub-domain ECUs. Instead, they mainly pre-process data and transmit it to a more powerful central computing unit for further processing. Zonal architectures are regarded as the most efficient electrical and electronic architecture [7].

2.2 Automotive Ethernet and Time-Sensitive Networks

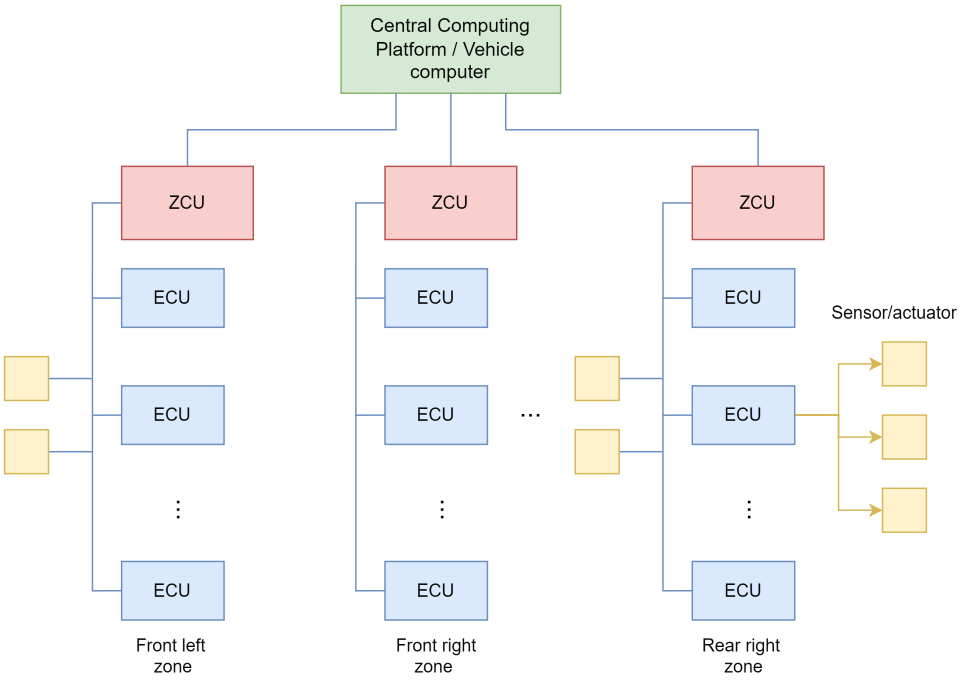


Fig. 2.5: Typical zone-oriented EEA

Automotive Ethernet

The automotive industry is adopting Ethernet technologies to handle the increasing communication bandwidth required by driver assistance and infotainment systems. Modern EEA architectures are organized hierarchically, with Ethernet as the backbone that connects domain controllers or links zonal controllers to the central computing platform [13]. 'Automotive Ethernet' refers to any Ethernet-based communication within the in-vehicle network, which may include different Ethernet standards such as 10BASE-TX and 100BASE-TX. For instance, modern vehicles rely on high-speed Ethernet for infotainment networks but on legacy networks such as CAN for control functions, especially time-critical ones. Consequently, 10BASE-T1S [14] Ethernet was introduced as a low bandwidth, low latency, multi-drop, Ethernet bus technology to replace legacy communication protocols and unify the in-vehicle network. This development is driven by implementing the zonal architecture, which aims to provide Ethernet connectivity to edge sensors and actuators [15].

Time-sensitive networks

Time-sensitive networking refers to the set of standards that enable reliable, bounded low-latency communication within the vehicle. TSN works with other IEEE technologies to provide

timing synchronization, reduce jitter, and traffic scheduling, guaranteeing that bounded low latency messages meet their deadlines [16]. TSN standards include protocols such as:

- IEEE 802.1AS that provides timing synchronization for time-sensitive applications, such as audio, video, and time-sensitive control, across the network [17].
- IEEE 802.1Qat SRP (Stream Reservation Protocol) that specifies mechanisms to reserve network resources for specific traffic streams [18].

2.3 In-vehicle communication protocols

In the following section, we provide an overview of two in-vehicle communication protocols: IEEE1722 and SOME/IP

2.3.1 AVTP (Audio Video Transport Protocol)

AVB (Audio/Video Bridging) refers to a set of specifications that enable time-synchronized, low-latency streaming services over IEEE 802 networks. It includes protocols such as gPTP (generalized Precision Time Protocol) described in IEEE Std 802.1AS, as well as protocols such as SRP (Stream Reservation Protocol) and FQTSS (Forwarding and Queuing Enhancements for Time-Sensitive Streams). AVTP (Audio Video Transport Protocol), which is defined by IEEE Std 1722, takes advantage of these standards to define a layer 2 transport protocol used for transmitting audio, video streams, and control data over a TSN network. An AVTP stream is made up of a talker and one or more listeners. The protocol provides methods to transport data and timing information so that audio, video, or control data sent by the Talker can be reproduced on the Listener's side. The protocol was originally designed to transport audio/video streams and their synchronization clocks. However, AVTP's scope was expanded in its 2016 spec [19] to include data formats for the serialization of automotive field buses, such as CAN, LIN, and FlexRay, by introducing a new control format that supports the transmission of various control messages over TSN. The next amendment to the protocol, proposed by IEEE P1722b working group [20], is going to be released in early 2025 and will further expand the protocol to include additional control formats, including I2C and SPI, which are relevant to the automotive industry. IEEE1722 is a Layer 2 transport protocol with its own EtherType ($0x22f0$) but also provides UDP/IP encapsulation. The standard mainly focuses on the definition of data formats and encapsulation for streaming applications

An AVTP frame presented in Figure 2.6 consists of the following components :

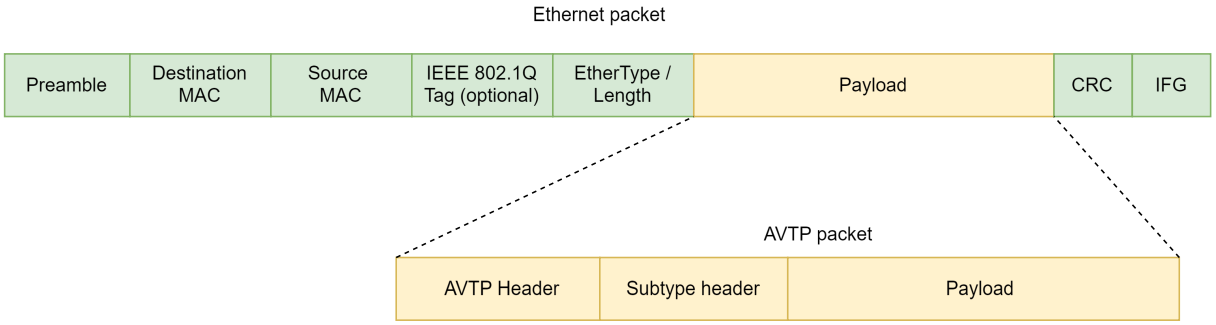


Fig. 2.6: Ethernet packet with AVTP frame as payload

- **Header:** contains information on the formatting of the frame. AVTP defines four different types of headers:
  1. **Common Header:** contains basic fields shared by all formats, such as the subtype and version.
  2. **Common Control Header:** expands the Common Header to all the fields shared by streaming formats.
  3. **Common Stream Header:** expands the Common Header to all the fields shared by control formats.
  4. **Alternative Header:** defined for formats that do not fall into the two previous categories.
- **Subtype Header:** Identifies the subtype. Subtypes define the formatting and purpose of the payload. Each subtype has unique encapsulation and processing rules. One example is the AAF subtype, which is a streaming subtype that provides a mechanism for transporting audio over the network.
- **Payload:** contains actual media/data content such as a compressed audio stream or ACF messages.

AVTP defines the ACF (AVTP Control Format) frame for transmitting control signals. It relies on two types of headers: time-synchronous and non-time-synchronous. The time-synchronous header, referred to as TSCF (Time-Synchronous Control Format) header in Figure 2.8, includes an extra presentation time field used to achieve timing synchronization between the Talker and Listener(s) application. Presentation time represents the gPTP time on which the data contained in a AVTP frame will be available to the Listener (i.e., the Listener starts processing the

data when the presentation time is reached). Presentation time accounts for the variability of transit times between the Talker and Listener(s). The other header type, the NTSCF (Non-Time-Synchronous Control Format) in Figure 2.7, does not include this field.

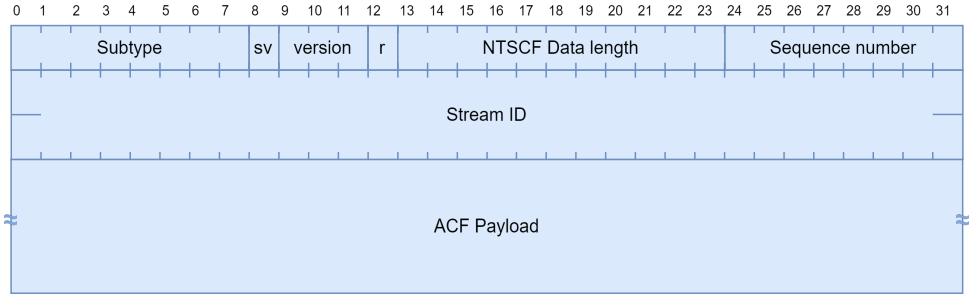


Fig. 2.7: Non-Time-Synchronous Control Format structure

An NTSCF frame includes the following fields :

- **version:** specifies the version of the format.
- **sv:** specifies whether the stream ID is valid.
- **NTSCF data length:** specifies the ACF payload length in octets.
- **sequence number:** sets the sequence number of the AVTP frames in the stream, which is incremented by one on every transaction.
- **stream ID:** used for stream identification/traffic management (defined IEEE Std. 802.1Q).
- **ACF payload:** contains one or more ACF messages.

While TSCF frame includes the following fields that were previously not defined:

- **stream data length:** length in octets of the ACF payload

An example of control messages is the GBB (Generic Byte Bus) message that allows the transport of SPI, I2C, and similar byte-oriented buses over the network. It is important to note that GBB messages are not included in IEEE1722-2016 but will be introduced in the next amendment to the protocol [21]. The proposed structure of a GBB message is presented in Figure 2.9. The message includes the following relevant fields:

- **ACF Message type:** Defines the message type contained in the payload of the ACF frame. ACF supports multiple message types for various control applications, including CAN, LIN, and SENSOR messages.

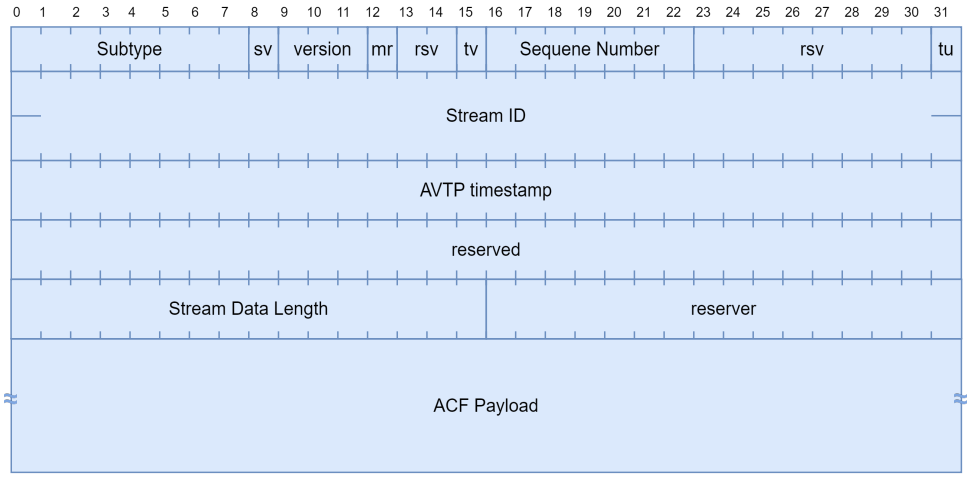


Fig. 2.8: Time-Synchronous Control Format structure

- **ACF Message length:** Indicates the length of the message in quadlets (multiples of 4 Bytes)
- **pad:** Defines the padding length at the end of the messages in octets. ACF frames must end at the boundary of the quadlets.
- **mtv:** Indicates that the message timestamp is valid/
- **Byte bus ID:** Provides an identifier for the byte bus associated with the message. Implementation of this field is application-specific.
- **Message timestamp:** Acquisition time in nanoseconds of the data contained in the control message (i.e., the time on which the Talker generated the message).
- **evt:** Reserved for application specific events.
- **Transaction number:** Indicates the sequence number of the GBB messages in a stream. Its value is incremented by one with every message. It could be used by the Listener to detect message loss and to associate the response with the original request.
- **op:** Specifies whether the operation is a read or write.
- **rsp:** Indicates if the operation is a request or response.
- **err:** Indicates whether an error occured.
- **ms:** Signals to the Listener that the payload is segmented into multiple messages.

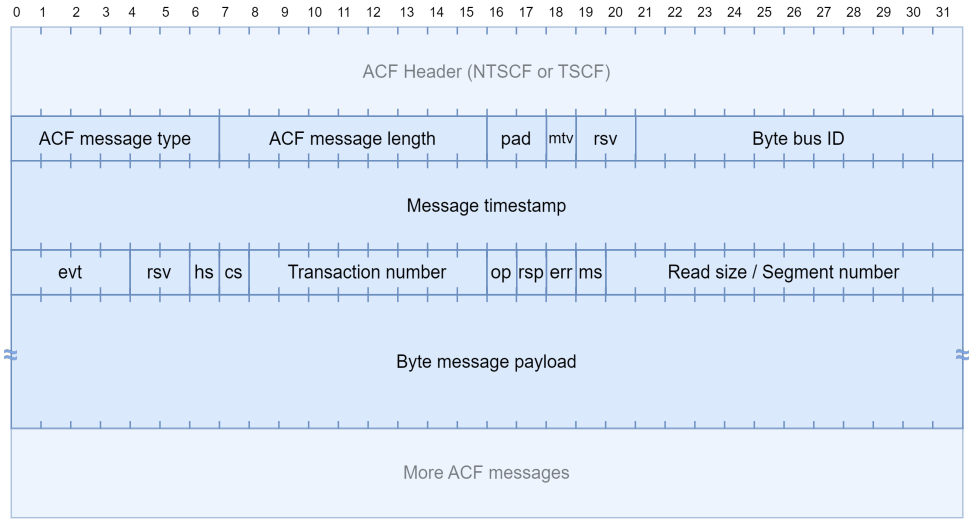


Fig. 2.9: Generic Byte Bus message structure

- **Read size / Segment number:** Specifies the segment number for the fragmented message or the data size to read from the bus in case of a read operation.

To recap, IEEE1722 already provides efficient transport methods for legacy communication protocols like CAN and LIN via Ethernet-based vehicle networks. A planned amendment aims to support additional control signals in the future. The protocol is also integrated into AUTOSAR specifications [22] and could be potentially implemented in hardware.

2.3.2 SOME/IP

SOME/IP (Scalable service-Oriented MiddlewarE over IP) is an automotive middleware and communication protocol for control messages. It supports remote procedure calls and event notifications and defines a custom serialization format for on-wire representation. Additionally, it includes a service discovery mechanism that enables dynamic identification of functionalities. It is designed to work on various devices and operating system platforms, from basic devices like cameras to more advanced ones like head units. It serves as a replacement for MOST protocol in infotainment applications and also replaces traditional CAN in some use cases [23].

SOME/IP specification is defined by AUTOSAR [24]. The protocol enables service-oriented communication over the network. Each service provides a list of functionalities that combine zero or multiple events, methods, and fields. The Client subscribes to the services that it needs to interact with. The middleware offers a custom API to provide an abstraction layer for the user applications from the underlying platform's network stack (Figure 2.10). Applications only need

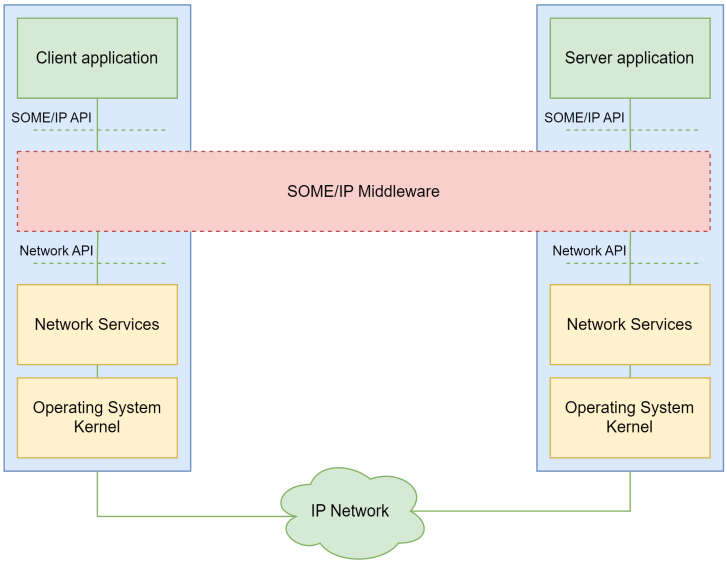


Fig. 2.10: SOME/IP middleware

to know the IDs of the clients or services they're communicating with; the middleware manages the low-level network transport details.

Figure 2.11 presents the supported communication patterns by SOME/IP:

- **Request-Response:** The Client sends a request message, which the server receives and answers.
- **Fire-and-Forget:** The Client sends a request without expecting a response from the server. This applies to error messages, too, as the server will not send any form of response, and error handling happens locally at the Client.
- **Notfication Events:** Follows a publish/subscribe pattern on which the Client subscribes to a specific event. The server sends an update once that event occurs. The update could be either an updated value or a flag for the event that occurred. Event notifications are useful for scenarios when the Client needs to poll a value (e.g., periodically read temperature sensor value) or needs to be notified when a value changes or a condition is reached (e.g., temperature increased over a certain limit).
- **Fields:** Provides a combination of getters, setters, and event notifications. The Client subscribes to a field, which represents a status and has a valid value. The Client can send a message to read or update the field, and the server notifies the Client when its value changes.

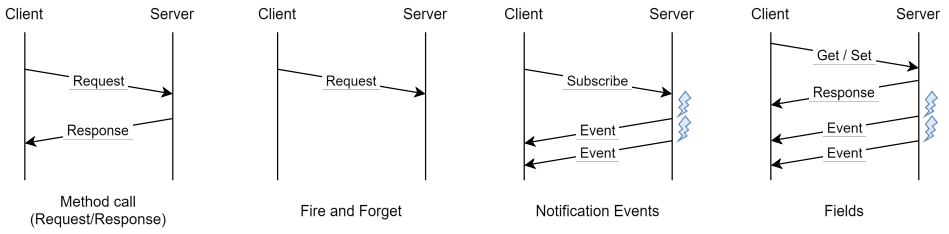


Fig. 2.11: SOME/IP communication patterns

SOME/IP's discovery mechanism is defined in the SOME/IP-SD (Scalable service-Oriented MiddlewarE over IP - Service Discovery) specification [25]. Service discovery establishes a communication path during runtime, which enables the flexible design of Talker and Listener applications as they are no longer required to know information on who will receive or send the desired data. In other words, a server can offer a service, and the Client finds and subscribes to it without statically knowing its location in the network. Figure 2.12 showcases SOME/IP's service discovery mechanism. Clients and Services broadcast 'find' and 'offer' messages that allow them to locate each other and establish communication. Service discovery relies on the Service and Instance IDs of each service. The service ID distinguishes between services, while the instance ID distinguishes between different instances of the same service.

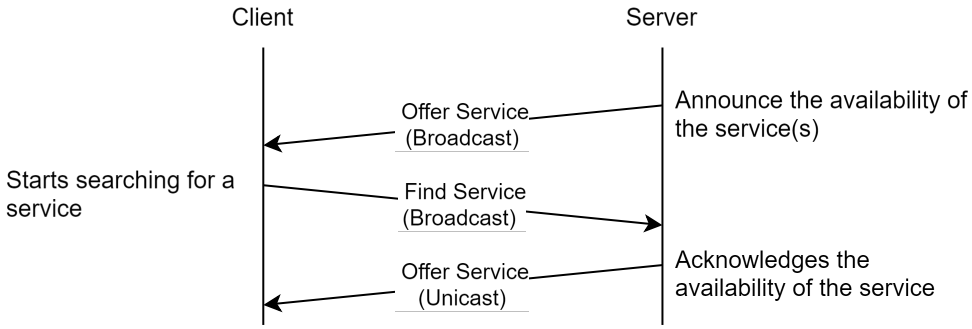
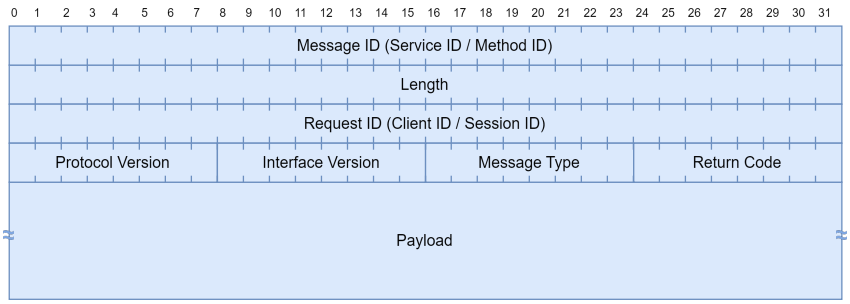


Fig. 2.12: SOME/IP service discovery

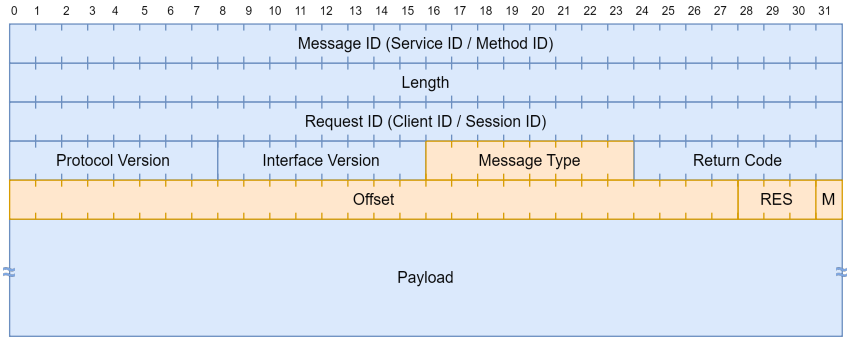
SOME/IP defines a custom serialization format that describes how data fields are encoded into a byte stream for transport over the network. SOME/IP messages can either be unreliable, transported over UDP, or reliable, transported over TCP. Unreliable messages are limited by the maximum transmission unit of a UDP packet. For this reason, SOME/IP-TP (Scalable service-Oriented MiddlewarE over IP - Transport Protocol) specification [26] defines a mechanism to segment unreliable SOME/IP packets that do not fit into a single UDP packet. Figure 2.13 com-

pares the regular and transport protocol headers of a SOME/IP message. The message includes the following fields:

- **Message ID:** composed of two fields, the Service ID and the Method ID. The Method ID distinguishes between methods and/or events within the service.
- **Length:** Length in Bytes of the remaining part of the message.
- **Request ID:** is used to identify a request-response pair to distinguish between multiple calls from the same message. It consists of two fields: the client ID and the session ID. The client ID is mapped to the Client that issued the request, while the session ID is an identifier incremented with every request.
- **Protocol Version:** Identifies the header format version.
- **Interface Version:** Identifier the interface version of the service.
- **Message Type:** Selects message type, which includes requests, fire-and-forget requests, responses, and error messages.
- **Return Code:** Indicates if the request was successful. Includes one of the pre-defined or custom error codes.



(a) Regular



(b) Transport Protocol

Fig. 2.13: SOME/IP header formats

3 Framework for a Remote Control Protocol

This chapter establishes a concept of remote control protocols. Section 3.1 defines the system model of a remote control system. Section 3.2 explores some use cases for remote control. Followed by Section 3.3, which extracts the core requirements for a remote control protocol, and finally Section 3.4 shines the light on ongoing RCP standardization and their role in defining the protocol.

3.1 System model

A remote control system enables efficient control of edge nodes over the network. It minimizes the software running on these nodes, thus reducing their hardware complexity and cost while allowing the further concentration of software in the central computing unit(s) of the vehicle. The standardization makes edge nodes commodity IP (pre-designed intellectual properties). The system consists of a set of nodes interconnected by a switched or routed network. One node exposes interfaces, such as SPI, I2C, and GPIO, that other nodes can access transpar-

ently over the network. A node can either act as a server (exposing an interface) or a client (controlling an interface), allowing the system to adopt a variety of architectures. A client/server architecture is one possible manifestation of a remote control system.

The system's operation can be defined by an RCP (remote control protocol) that defines the system initialization, configuration of remote interfaces, diagnostics, communication between nodes, and error handling. Therefore, a remote control protocol defines a standardized method for accessing and controlling interfaces on remotely connected nodes. It governs the data formats and rules of interaction between nodes in the system.

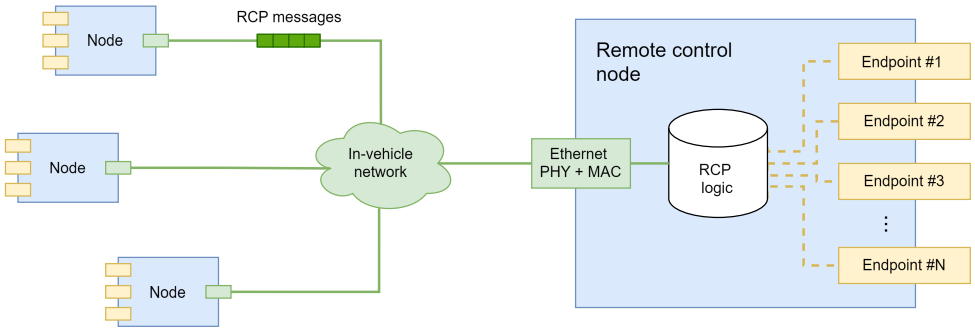


Fig. 3.1: Remote control system architecture

Figure 3.1 presents a high-level overview of the remote control system architecture. Access to a remote interface is achieved through an endpoint that exposes a set of operations defined by the interface. These operations include:

- Reading the interfaces's current state (e.g., monitoring the interface for errors or failures)
- Configuring the interface (e.g., configuring the clock speed of an I2C interface - changing between standard and fast I2C modes)
- Handling the interface's core functionality (e.g. performing I2C read or write operations on the device(s) connected to the interface)

These operations can be described by remote control messages, which are carried over the network by a remote control frame between the RCP client and the RCP node. The rest of this section defines the components of a remote control system in more detail:

- **Remote Control Node:**

  The adoption of the remote control concept leads to the introduction of a new category of edge nodes known as the remote control nodes (or RCP nodes). Edge nodes are

no longer required to abstract the functionality of connected sensors and actuators. Instead, they are required to act as gateways that enable the exchange of data/commands between the central ECU and sensors/actuators. As a result, an RCP node replaces the traditional edge node. The exact definition of the internal architecture of the RCP node is still an open question. Addressing this question mainly depends on the specific requirements for RCP and how they are implemented. For example, a node can rely entirely on a custom hardware implementation to perform its function (MCU-less). However, some more complex requirements may necessitate using a basic MCU with a lightweight software stack.

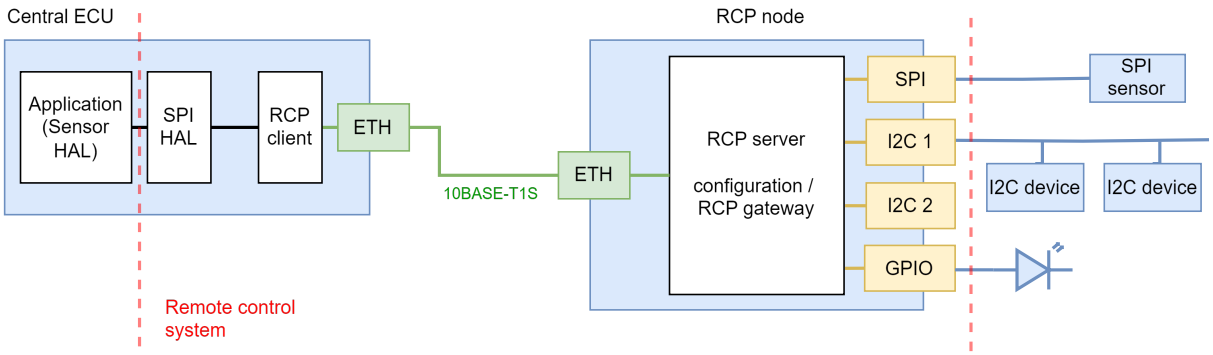


Fig. 3.2: example of a basic remote control scenario

Having not explored the requirements for remote control yet, this section focuses on defining the high-level concept of an RCP node. Figure 3.2 shows an example of a basic remote control scenario. At a minimum, an RCP node may include at least one peripheral such as I2C, SPI, GPIO, and at least one Ethernet interface, alongside a compute unit or hardware logic that handles the operation of the node. The remote control protocol manages the peripherals and the node facilitating access to the devices attached to these peripherals. To simplify its implementation, the RCP node should not require any knowledge of the operation of these devices. The application solely handles the peripheral's operation, and the RCP node acts as a gateway between the application and the peripheral. Additionally, the operation of the remote control system should be transparent to the application. The client could provide an abstraction layer that allows the application to access the interfaces as if they are local, which makes it easier to port legacy applications to the new system.

- **Remote Control Endpoint:**

By definition, A device of a software module connected to a computer network is considered an endpoint of ongoing data exchange with other devices on the network [27]. In this context, a remotely controlled interface can be defined as an endpoint of an ongoing data or command exchange with the remote control client. In other words, the endpoint is an access point for controlling the devices connected to the interface. Since an RCP node may host multiple interfaces, the node should provide more than one endpoint that the client can independently address. Let us consider an SPI peripheral as an example of a remote control endpoint. The endpoint should expose a set of status parameters and flags associated with the interface. For an SPI interface, this would include error flags to indicate an overrun condition error, which would be valuable for the application when managing faults. It is essential to point out that implementing a remote control node should be generic and independent of the devices to which it is connected. For instance, the same node can connect to two different SPI devices on the same bus. For this reason, the client would need to modify the peripheral's internal configuration to work with the target device; this would involve setting parameters such as the SPI clock frequency, clock polarity and phase, data order, chip select management, and data frame size. Finally, the endpoint should allow the client to execute the peripheral's core functionality, such as sending SPI write or read commands.

- **Remote Control Messages:**

  In the context of RCP, which is a communication protocol, a remote control message is a piece of data or information exchanged between the RCP client and the RCP endpoint or node. A message describes an operation requested by the client, or a notification by raised by the server. For instance, messages can be classified into queries, configuration, and control. A query message enables the client to read the current state of the peripheral. A configuration message allows the client to change the configuration of the peripheral, and a control message triggers the core functionality of the interface. Messages may also include reporting messages such as acknowledgments and error reports. Each message holds an exact meaning and may or may not elicit a response from the endpoint. For an SPI interface, the messages include operations such as SetClockFrequency, SetMode, ReadNByte, WriteNByte, and others. The exact structure and format of these messages will be defined by the remote control protocol

- **Remote Control Frame:**

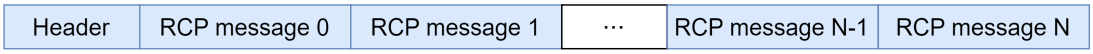


Fig. 3.3: Remote control frame structure

A remote control frame refers to the structured set of messages sent over the network. Figure 3.3 shows the structure of a remote control frame. It includes a header that contains information about the message and network transport (such as the network address) and a payload that encapsulates one or more remote control messages.

- **Network:**

  The remote control network serves as the medium on which different nodes exchange RCP messages. In the context of automotive applications, this refers to the in-vehicle network, which may be a switched or routed Ethernet network. The RCP protocol requires a transport mechanism to deliver its frames between nodes on the network.

Ideally, the protocol's transport mechanism should be flexible enough to operate on top of any network layer. For instance, an RCP frame could be encapsulated in a UDP packet, or the same frame could be encapsulated in a raw Ethernet frame, resulting in a remote control protocol that is independent of the underlying transport mechanism. However, depending on the system and network requirements, the remote control protocol could offload certain functionality to the transport mechanism, simplifying the implementation of its logic. For instance, if the underlying transport mechanism supports segmentation (splitting a frame into smaller chunks when larger than the minimum transfer unit size), the remote control protocol would not need to handle this task. In essence, creating a 'Swiss knife' solution for the given problem is not always necessary. Instead, it is essential to examine the requirements for a remote control system in more detail, which acts as a base for the protocol definition and the selection of its transport mechanism. This thesis addresses the issue of selecting a transport mechanism by evaluating potential candidates of underlying transport protocols and how they would align with the RCP model discussed in this section. To achieve this, we must first define a set of requirements and identify the challenges associated with implementing RCP. In the next section, we will explore some use cases for RCP, which will later act as a basis for defining the requirements for the protocol and its transport mechanism.

3.2 Use-cases

Vehicles usually contain a wide range of peripherals controlled by interfaces like GPIO, I2C, SPI, and UART. For example, GPIO interfaces manage basic functions such as turning on indicators or activating windshield wipers. SPI and I2C interfaces are also used to read data from transceiver modules, such as tire temperature and pressure sensors. Other applications may include streaming video or audio from interior and exterior cameras. Such peripherals usually handle both non-critical applications, like adjusting exterior mirrors, and critical applications, such as controlling brake-by-wire systems. In this section, we will examine some scenarios in which a remote control mechanism can be used to adapt certain applications to work in a centralized EEA. Then, we will go deeper into the interface level and discuss the requirements for controlling remote I2C and SPI peripherals.

3.2.1 Use-case 1: Controlling Ambient lighting Systems

Car makers are experimenting with dynamic ambient lighting as an innovative solution to increasing the driver's situational awareness. Adding dynamic interior lights to a vehicle's cabin adds a new dimension to communication between the car and its occupants. Animated optical effects can be used to communicate crucial information to the driver. For example, a light strip at the top of the door panel can warn a passenger of potential danger in the path of their door when they attempt to open it. Ambient lighting can also contribute to the overall interior aesthetics of the cabin [28]. Studies have shown that the driver's performance is enhanced by using ambient lighting as an emotional feedback mechanism (e.g., switching the interior light color to orange to increase the driver's alertness in certain driving conditions [29]).

To align with the centralization trend, it is essential to design a flexible, scalable, and easy-to-maintain and update ambient lighting system. Customization is an essential requirement for such a system, as drivers would likely have different preferences as to how these animations work. Manufacturers can allow customers to select from a predefined set of functions and animation patterns or even design custom ones. This flexibility can be achieved by moving all the lighting software to the central ECU and using a remote control mechanism to control the lighting elements. A proof of concept of such a system is presented in [30]. Figure 3.4 presents its system architecture. The system uses custom gateways to translate communication over 10Base-T1S Ethernet to ILaS. ILaS, which stands for ISELED Light and Sensor network, is a smart RGB LED solution from Microchip to expand the use of the ISELED protocol to other

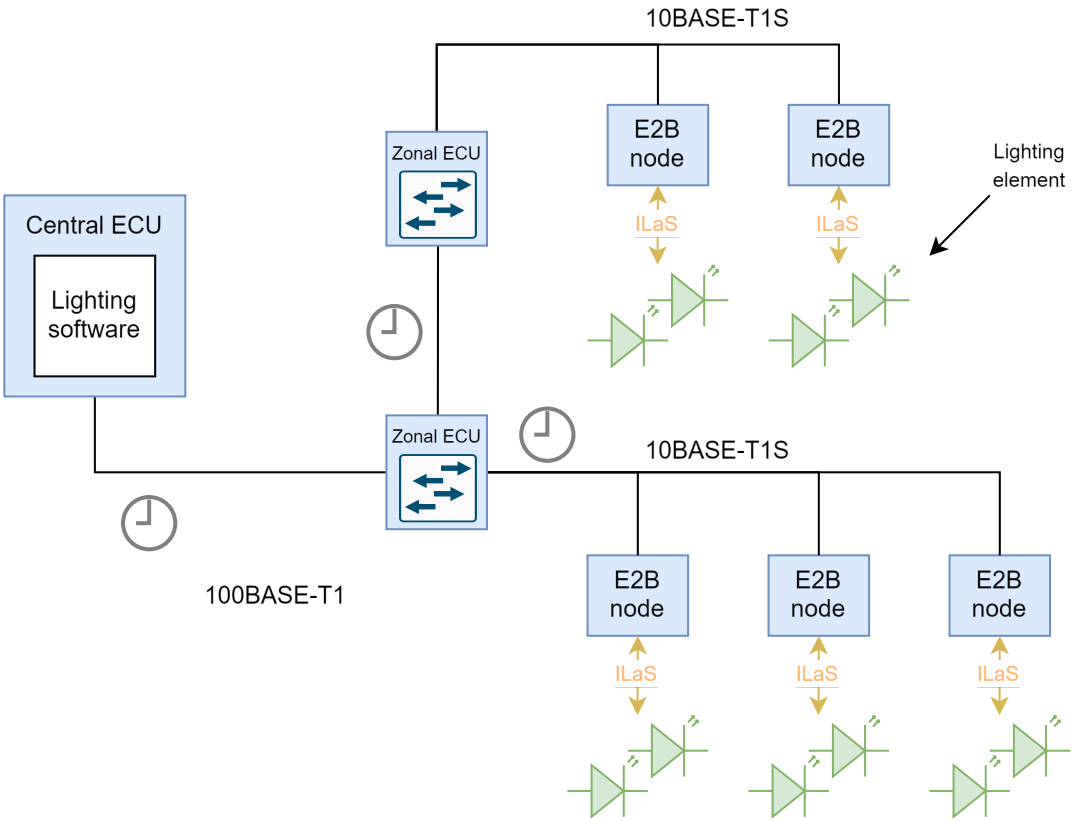


Fig. 3.4: Ambient lighting system architecture (adapted from [30])

components, such as sensors and actuators, while ISELED is a protocol designed for dynamic ambient and functional lighting control applications. It simplifies color design and connectivity for managing a high number of LEDs [31].

The lighting software on the central ECU uses the switched network to remotely control the ILaS interfaces, which are driving the interior ambient lighting strips. This is done using propriety hardware-based Ethernet edge nodes (i.e., remote control nodes) from Analog Devices. Particularly, the AD3300X family of edge nodes [32] that includes a 10BASE-T1S E2B (Ethernet to the Edge Bus) transceiver which supports simultaneous operation of several standard sensor/actuator interfaces, including SPI, I2C, UART, PWM, GPIO, Flexible I/O, and bridge to LIN, in addition to a bridge to ILAS and ISELED interfaces. E2B edge nodes were used to eliminate the requirement of dedicated MCUs on the edge. Consequently, performing an OTA, update would only be required to update the lighting control application on the central ECU. Using dedicated hardware also ensures deterministic control of the remote peripherals as it eliminates the unpredictability introduced by the software stack of an MCU. The system also provides con-

nectivity diagnostics to the central ECU and offers a higher level of security due to the reduced attack vector of hardware-based implementations.

Two key requirements presented in [30] for their proof-of-concept ambient lighting remote control system are the synchronization of lighting animations and cross-application integration. Lighting animations must be stable, achieving a minimal jitter (e.g., updating lighting elements at a consistent and perceivable refresh rate with jitter below 2 ms), and accurate (e.g. executing an animation precisely after 500ms second with a timing precision of approximately 20 ms). Additionally, with the distribution of the lighting elements and control signals across the network, the applications generating these animations should communicate within a certain delay threshold with other system functions to ensure consistent operation (e.g., The delay between turning on the hazard indicator and starting its associated lighting animation should not exceed 150ms to ensure a smooth user experience). With these requirements in mind, it is clear that in some scenarios, the system has to meet strict latency requirements. It is also evident that the system must implement a time synchronization mechanism to align all of its actors on a unified timebase to reduce jitter and improve overall stability. This could also be achieved by offloading some basic functions to the remote control logic. For instance, the remote control nodes can offer the ability to schedule operations for future execution.

3.2.2 Use-case 2: Camera streaming and control

With automotive networks shifting to an Ethernet backbone, data from cameras used in ADAS features, such as lane departure warning, traffic sign recognition, night vision, and bird's-eye view, will be transmitted via Ethernet. Protocols such as AVTP are going to replace existing ones like the MOST protocol, which is currently used in such applications [33]. Camera-based ADAS currently use point-to-point connections to ECUs through Voltage Differential Signaling (LVDS) or newer interfaces like MIPI-CSI. However, Ethernet connections to the zonal ECUs are set to replace the traditional point-to-point links. In this case, Ethernet camera gateways could tunnel point-to-point camera streams from the edge to the central ECU over ethernet [34].

Besides video streaming, automotive camera modules typically rely on other serial protocols for initialization/configuration. For example, the TIDA-01130 camera module from Texas Instruments [35] includes a MIPI CSI-2 interface for raw video streaming, in addition to an I2C interface to initialize the module. Figure 3.5 showcases an example scenario of integrating a camera module into a remote control system. In this case, the RCP node controls two interfaces

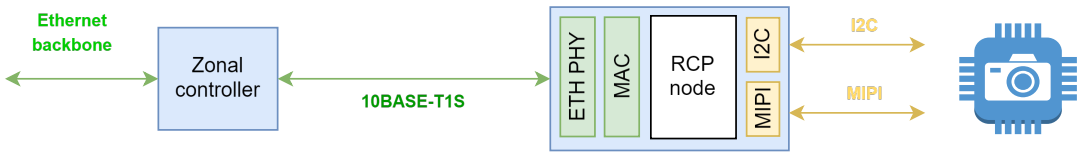


Fig. 3.5: Example scenario for remotely controlling a camera module

associated with the same camera module. If enough bandwidth is available, an RCP node could potentially connect to more camera modules, requiring additional interfaces on the node.

In-vehicle video streams, particularly the ones used for critical systems, such as pedestrian detection and collision avoidance, mandate strict real-time requirements. Performing object detection on the central ECU poses some concerns about the bandwidth and latency demands from the remote nodes. This approach requires the transmission of raw or pre-processed video streams to the central ECU, which can be bottlenecked by bandwidth limitations and introduce latency. Experimental results [36] highlight a trade-off between the accuracy of object detection models, their speed, and the required bandwidth. Object detection models are more effective with higher bitrate video streams. Also, detecting objects earlier gives the vehicle enough time to respond to potential hazards. Therefore, constrained communication between the camera module and the detection software could compromise or reduce the effectiveness of object detection, which implies that the remote control system must ensure minimal latency and bandwidth overhead on the video stream.

Initializing the camera module via I2C provides the advantage of managing multiple modules using a single I2C interface. For example, two camera modules located near each other can be initialized over the same I2C bus, provided that their I2C interfaces have distinct addresses. The next section/use case dives deeper into the interface level, exploring the challenges of remotely controlling an I2C peripheral.

3.2.3 Use-case 3: Controlling I2C peripherals

The Inter-Integrated Circuit bus [37], commonly known as the I2C bus, is a simple bidirectional 2-wire bus designed by Phillips Semiconductors (now NXP Semiconductors) to enable communication between Integrated circuits and microcontrollers. It is a multi-controller, multi-receiver, synchronous, bidirectional, half-duplex serial communication bus. It operates over two lines: a Serial Data Line (SDL) for transmitting data and a Serial Clock Line (SCL) for synchronizing the communication. Figure 3.6 shows an example of an I2C bus application on which a microcon-

troller communicates with different types of peripherals over I2C. The SDA and SCL lines are

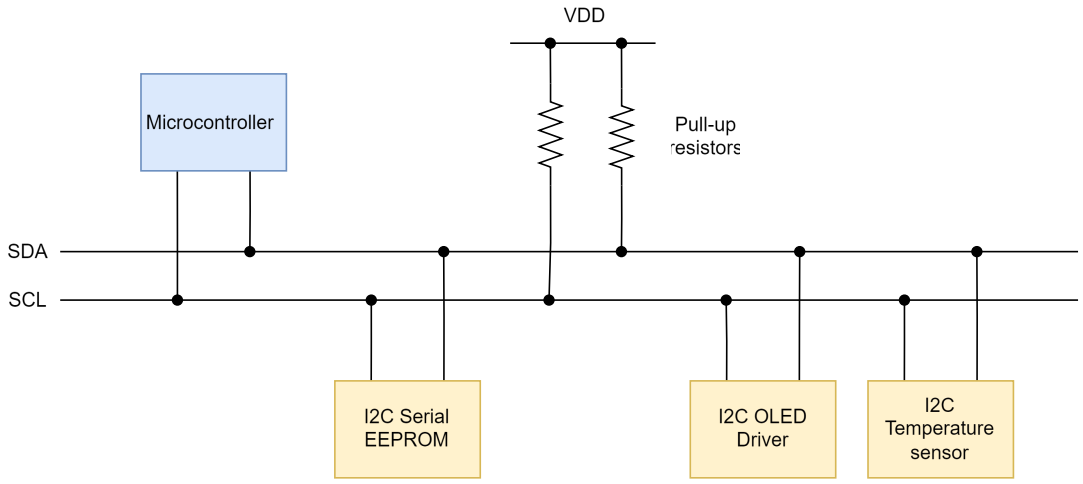


Fig. 3.6: Example of I2C bus application

HIGH by default and need to be connected to a positive supply voltage, typically done using pull-up resistors. Devices connected to the bus pull the voltage LOW to transmit a 0 bit and leave the lines HIGH to transmit a 1 bit. Each device is identified by a unique address and can both transmit (write) or receive (read) data—devices on the bus act either as controllers or receivers. The controllers initiate data transfers with the designated receivers. The bus supports a multi-controller configuration, and It implements collision detection and arbitration mechanisms to prevent data corruption if two or more controllers simultaneously initiate data transfer. Figure 3.7 presents a complete I2C data transfer. All I2C transactions begin with a START (S)

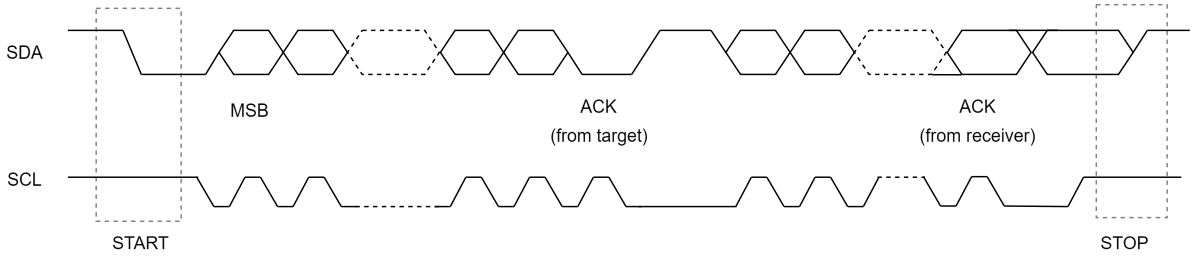


Fig. 3.7: I2C data transfer

condition and end with a STOP (P) condition. Additionally, the controller, which is responsible for generating these conditions, has the option to keep hold of the bus by generating a repeated START (Sr) condition in between transactions. The start condition is then followed by the target address and an extra bit to indicate the direction of the transaction - a 'zero' indicates

data transmission from the controller to the receiver while a 'one' indicates a data request. The receiver acknowledges that it is ready for the transfer by pulling the data line low. Data is then transmitted sequentially, 1 byte at a time, with each byte followed by an acknowledgment (ACK) or a non-acknowledgment (NACK). In a write operation, the receiver acknowledges the receipt of each byte, while in a read operation, the controller sends the acknowledgment. The controller determines when to end the transaction when the required number of bytes sent/received is reached.

Receivers can pause the transaction by holding the SCL line low after receiving and acknowledging a byte. The transaction is halted until the SCL line is released. This process is known as clock stretching. The target devices can force the controller to wait until the device finishes processing the received or requested data. In this section, we will briefly describe the properties of the I2C bus and explore the challenges and requirements for remotely controlling an I2C peripheral.

Peripheral Configuration

The I2C bus specification [37] defines a set of mandatory and optional parameters. Exploring all the possible configuration parameters of an I2C bus can be complex and is not the main focus of this work. For this reason, we will only consider a subset of these parameters. As an example, we will examine the I2C peripheral definition from an STM32H7 microcontroller [38]. Table 3.1 outlines the main configuration parameters of this peripheral. These parameters can be classified into two groups: initialization parameters and runtime parameters. Initialization parameters are the parameters that are set before a peripheral starts its operation, while runtime parameters are the parameters that change during operation based on the requirements of the next transaction. For instance, to initialize the STM32H7's I2C peripheral, the user must configure the clock frequency, analog/digital filters, timing parameters, and clock stretching options before enabling the peripheral. On the other hand, the user has to set the addressing mode every time a new transaction is started (i.e., Receivers with different address lengths can coexist on the same bus, it is up to the controller to select what addressing mode to use when initiating a transaction). RCP should define both a mechanism to set the initial configuration parameters and update the runtime settings of the peripheral. A key challenge is determining the abstraction level provided by the remote control protocol. Is RCP expected to provide direct access to the internal registers of the peripheral, or are the inner workings of the peripheral solely handled by the RCP node? Constantly sending requests over the network to update

Tab. 3.1: Configuration parameters of an I2C peripheral

Parameter	Applies to	Description
Peripheral enable	Both	Enables the peripheral
Mode (Role)	Both	Defines whether the device is a controller or a receiver
Clock frequency	Controller	Defines I2C clock speed (e.g. 100KHz, 400KHz, 1MHz, 3.4MHz)
Clock stretching	Receiver	Enables clock stretching (e.g., allows the receiver to hold the SCL line low to slow down the communication)
Addressing mode	Both	Selects between using 7-bit and 10-bit addresses
Address	Receiver	Sets own device address
Dual addressing mode	Receiver	Enables the receiver to use multiple addresses
General call	Receiver	Enables the receiver to respond to general call address
ACK control	Both	Receiver sends an acknowledgment after the reception of each byte / Controller signals that it is ready to receive the next byte
Noise filters	Both	Sets up Analog/Digital noise filters on the SCL and the SDA lines
Setup time	Both	Sets the minimum time for SDA to remain stable before SCL rising edge
Hold time	Both	Sets the minimum time for SDA to be held stable after SCL falling edge
Software reset	Both	Releases SCL/SDA lines and cancels ongoing operation
Interrupts	Both	Enabes interrupts for event/error handling
DMA	Both	Transfer data to MCU using direct memory address

every register is slow and inefficient. Instead, a more practical approach is for RCP to provide a minimal level of abstraction. It could define a set of automatic operations that group related configuration parameters into a single RCP message. For instance, it could define a single initialization operation to set all the required parameters and enable the peripheral. A reasonable approach is to compare the abstraction provided by RCP to the level of abstraction provided by a microcontroller's hardware abstraction layer. For instance, STM32's HAL [39] provides a $HAL\_I2C\_Init()$ method that reads configuration parameters such as clock frequency, duty cycle, and clock stretching mode, then internally handles the low-level initialization of the interface. RCP should follow a similar communication model on which the RCP stack sits between the peripheral's HAL on the controller node and the actual interface on the remote control node. RCP could define more specific initialization operations for certain runtime parameters in ad-

dition to offloading the transaction-specific parameters to the operations related to the actual transaction. Some settings related to the peripheral's operation can be entirely abstracted by the RCP node. For example, I2C peripherals have the option to perform direct memory access (DMA) to directly transfer data for a transaction from and to the memory, instead of waiting for the processor to manually write it to the peripheral. This type of interaction is not defined in the context of RCP and should not be managed by the protocol. DMA enables communication between the peripheral and the node's internal system components rather than between the peripheral and user applications.

Operations

The I2C bus supports two primary operations: transmit and receive. The Transmit operation governs data transfers from the controller to the receiver, while the Receive operation handles data transfers from the receiver to the controller. The protocol only defines a low-level transport mechanism for data exchange between the controller and the receiver. The application manages higher-level communication details, such as data formatting. For example, I2C-based memory devices define read and write operations. A read operation is performed by transmitting the data address to the memory controller, followed by a receive operation to read the data stored at that address. Similarly, a write operation consists of transmitting the address first, followed by transmitting the data to be stored at that address. The same concept is used when interfacing sensors/actuators; the controller transmits a command first, followed by a transmit or receive operation. While RCP is only responsible for handling the low-level operations, a possible optimization could involve defining these communication models as a single operation. RCP could implement the following operations: $transmit$, $receive$, and $transmit\_receive$. Each operation is targetted at a specific I2C bus and is defined by its direction (transmit/receive), target address, payload, and payload length or the length of data to read. In a $transmit$ operation, the controller sends a single RCP message containing the I2C payload. The RCP node receives the messages, extracts the payload, and performs the transmit operation locally. Then, based on the communication model between the controller and the I2C device, the RCP node can send back the operation status to the receiver or directly end the operation. In contrast, in a $transmit\_receive$ operation. The controller sends a message containing the transmitted data and the time it takes to read it. In this case, the RCP node must respond with the received data or an error message after locally executing both the transmit and the receive operations. A $receive$ operation, shown in Figure 3.8, is similar to this operation except that the controller

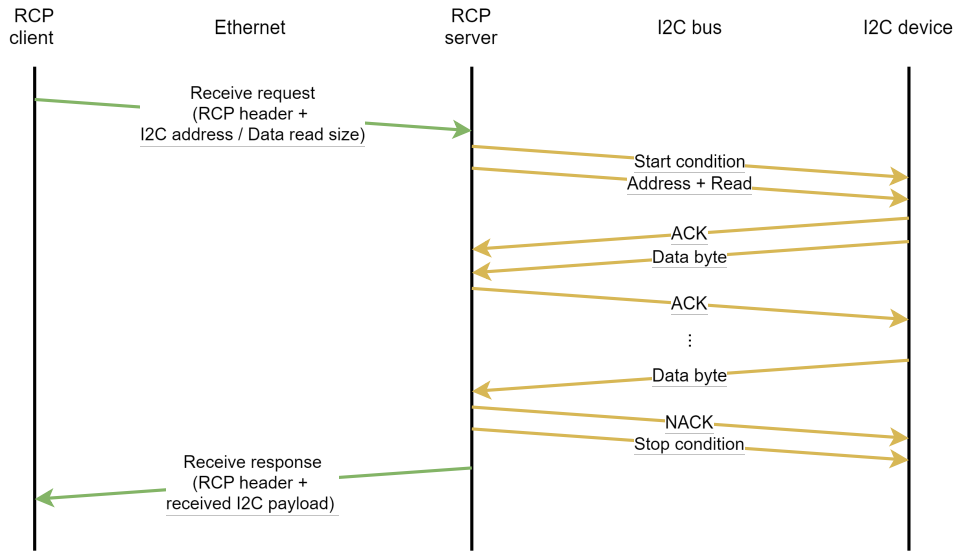


Fig. 3.8: I2C receive operation communication model

sends a message with no payload.

Error handling

I2C failures could be associated with improper configuration, transmission, or arbitration errors. STM32H7's I2C peripheral defines the following error conditions:

- Bus error: raised when an improper START or STOP condition is detected. It only occurs if the peripheral is involved in the communication (i.e., a controller or a receiver after the addressing phase).
- Arbitration loss: raised when another controller is holding the bus.
- Overrun/underrun error: raised when clock stretching is disabled and the receiver cannot process the received/transmitted data in time.

These errors occur during a transaction, meaning that the RCP node is not expected to send an unsolicited error message. Instead, it responds to a request already initiated by the controller. A bus error is caused by a signal integrity failure on the wire, potentially caused by misconfigured filters or pull-up resistors. However, the peripheral configuration is handled by the controller; the RCP node is only responsible for reporting this error to the controller. On the other hand, the RCP node could try to handle errors, such as the arbitration loss, locally. The delay between the initiation of the operation on the controller and its execution in the bus is affected by the

remote control protocol 's latency, which is further impacted by the network conditions. Addressing arbitration loss by re-attempting the transaction at the controller level would greatly increase the latency. Instead, the RCP node could retry the transmission locally within a defined timeout.

Example scenario: controlling a PWM/Servo driver

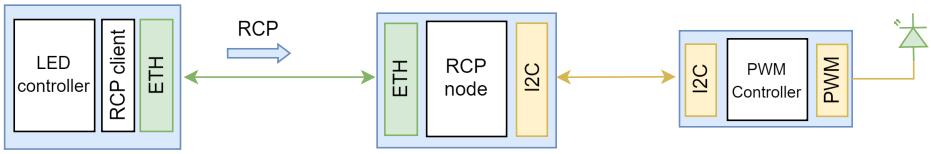


Fig. 3.9: Controlling an I2C-based device using RCP

In this section, we describe a basic example of controlling an I2C-based device. The system presented in Figure 3.9 consists of a PWM/Servo driver connected to one of the I2C peripherals of an RCP node. The driver is free-running. It uses an internal clock to generate a PWM signal on each of its output channels. The controller configures the device over I2C. To start its operation, the controller first verifies that the chip ID of the driver matches the predefined software ID, then initializes the driver by setting the PWM frequency and output status of each PWM channel. After initialization, the controller can update the duty cycle of individual channels by performing an I2C write operation on the driver's internal registers. We define a use case in which we blink an LED at a constant rate of 1 ms. This is achieved by continuously updating the PWM duty cycle of the signal controlling the LED between 0% and 100%. The LED's state is incompetent, meaning that the controller does not need to know the current state of the LED to send an update. This allows for fire-and-forget communication. The RCP node does not notify the controller when an update is successful. It can optionally reply with error messages if case of a failure for diagnostic purposes only. However, this doesn't apply to the initialization process, as the controller needs to verify the device's correct startup. Therefore, the remote control protocol should support both acknowledged and fire-and-forget operations. Another distinction could be made between synchronous and asynchronous operations. During start-up, all operations must be synchronous to follow the correct initialization order. For instance, the controller cannot update the PWM frequency before verifying that it is communicating with the correct device. In contrast, the controller can asynchronously send multiple update messages before receiving any response from the RCP mode. In this scenario, operation order is not

critical. If a message is lost, or if two messages are executed in the wrong order, the system operation may be disrupted, but it will not lead to complete failure.

3.2.4 Use-case 4: Controlling SPI peripherals

The SPI interface [40] is another serial bus designed for short-distance communication between integrated circuits. It was designed by Motorolla in the 1980s and has since become a de-facto standard. It is designed for point-to-point communication between a master (controller) and a slave (receiver). Multiple receivers are supported on a single bus. However, the specification does not implement an addressing mechanism; instead, a chip select (CS) signal is used to select the current receiver. It operates using four main signals:

- SCK (Serial clock): clock generated by the controller
- MOSI (Master Out Slave In): Data line for controller to receiver communication
- MISO (Master In Slave Out): Data line for the receiver to controller communication
- CS (Chip Select): Selects individual receiver

In addition to half-duplex communication, the SPI bus also supports full-duplex communication. Data is transferred simultaneously between controller and receiver over the MOSI and MISO lines. A message consists solely of the transferred data, without aknowledgement mechanism like in I2C. To start a transmission, the controller pulls the CS line low, and data is shifted between the internal registers of the controller and the receiver in both directions. Once the communication is done, the controller releases the chip select line.

Peripheral Configuration

We also examine some of the configuration parameters of an SPI interface from the STM32H7 microcontroller family, as shown in Table 3.2. Similarly, these parameters can be classified into initialization configuration parameters and runtime parameters. An SPI_init RCP operation can be used to initialize an SPI peripheral, and ideally, other operations could be defined to individually update each parameter and reset the peripheral to its default settings.

Peripheral operation

The SPI bus also supports two operations: $transmit$ and $receive$. Similar to the I2C bus, these operations can be combined into a $transmit\_receive$ operation where the controller issues

Tab. 3.2: Configuration parameters of an SPI peripheral

Parameter	Applies to	Description
Clock frequency	Controller	Sets the clock frequency
Frame size	Controller	Defines the number of bits transmitted per transaction
Clock polarity (CPOL)	Both	Determines the idle state of the clock
Clock phase (CPHA)	Both	Determines the data sampling edge of the clock
chip select (CS)	Controller	Sets the chip select signal
Bus mode	Both	Selects between half-duplex and full-duplex communication

a command or address to the receiver before transferring data. An SPI transaction is less complex as the controller only has to set the chip select signal, followed directly by transmitting the payload. Some SPI peripherals automatically manage the chip-select signals, while others do not. In this case, the application must manually control the chip-select signal using GPIO pins.

Additionally, standard SPI devices define an extra DC (data/command) signal, which is not part of the base SPI protocol. This signal is used to indicate whether the receiver must interpret the received transaction as a command or as data. RCP must handle the control of this signal either by making it a part of the SPI transaction itself or designating it as an extra GPIO operation. Handling it as an additional GPIO operation introduces some challenges. First, the order of the GPIO and SPI transactions must be consistent to ensure the correct operation of the device. Also, the timing behavior, namely the time between setting the DC signal and the start of the SPI transaction, must meet the timing requirements of the SPI device.

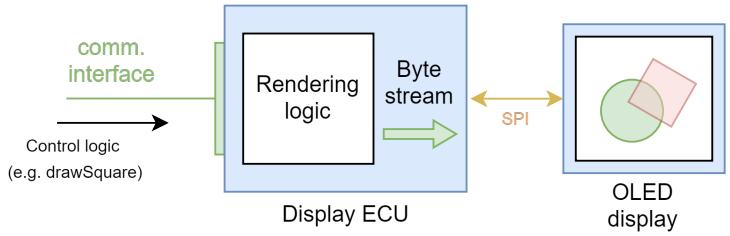
Example scenario: controlling an OLED display

In this section, we will describe a practical application of a remote control system on which a small OLED display is controlled remotely by managing the SPI peripheral it is connected to. The display is used to render basic animations, images, and patterns. Both the initialization and the streaming of raw frames to the display are done using the SPI bus. A frame is defined by a matrix where each element contains color information for one pixel. For this example, let us consider a 128x128 pixel, 16-bit depth grayscale OLED module. The module consists of an internal controller and an OLED panel. The controller is interfaced over SPI and is responsible for displaying the images on the panel. To keep it simple, let us assume that the display controller

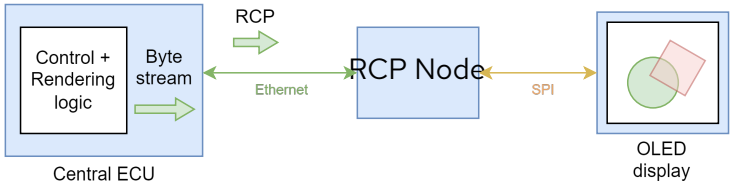
uses horizontal scanning to display the image. The frame is sent row-by-row as a byte stream starting from the top left corner of the display and finishing at the bottom right corner. For our display, where each pixel is represented by 4 bits, the total size of the frame is 8192 bytes ($\approx 8KB$).

In a traditional decentralized architecture, a dedicated ECU feeds the frame to the display over the SPI link. For this use case, we define a software function running on the ECU that is responsible for rendering the images to the display. We refer to this function as the rendering logic. The ECU receives high-level commands from other system components (other ECUs) containing details about the elements of the image to be rendered. For example, a square is described by its coordinates, width, height, and additional information about its color and outline. Similarly, a polygon is defined by a list of coordinates of its vertices. The final image is created by superimposing these objects on an empty frame. We refer to the application generating these commands as the control logic. In addition to generating the initial list of objects, the controller could also add or remove objects from the list, which implies that the rendering logic should maintain a record of the current state of the rendered objects. The application could also implement more complex features, such as associating an animation with a given object. For example, the control logic could define a fade-in animation from point x to point y for a specific square. The rendering logic is then responsible for correctly rendering this animation. The control logic only needs to know the commands for the definition of objects, their properties, and the different types of animations. Details related to the actual rendering and drawing [1] of the image, which is abstracted by the control logic. Implementing the same application in a remote control system implies that the rendering logic is moved to the zonal or central ECU. Figure 3.10 compares the OLED's use case system architecture in a traditional architecture to its implementation in a remote control system. The edge node is replaced by an RCP node and it no longer abstracts the rendering operations. Instead, the full frame is generated by the application and sent to the display using RCP. With RCP enabling transparent access to the SPI bus, the same rendering logic can be combined with the control logic to generate the frames. This approach introduces a few challenges. Compared to sending drawing commands, transmitting raw frames over the network demands additional bandwidth. Let us consider an example where we draw an 8x8 checkerboard pattern; the first implementation would require sending the de-

[1] In computer graphics, rendering is the process of generating an image from a 2D or 3D model. In this context, drawing refers to the stage on which the rendered image is displayed on the OLED display. For simplicity, we use the term "rendering"to describe the whole operation, including drawing



(a) Traditional



(b) RCP

Fig. 3.10: Comparison between different implementations of the OLED control use case

tails of 64 squares to the edge ECU. For example, on our display, a square can be described by the following parameters :

```
X coordinate     : 7 bits
Y coordinate     : 7 bits
Width          : 7 bits
Height         : 7 bits
Color          : 4 bits
Background opacity : 8 bits
Border color     : 4 bits
Border width     : 7 bits
Border opacity    : 8 bits
```

The object representing a square is approximately 8 bytes in size. The command to draw the frame is encoded using variable length encoding. Assuming the overhead from the encoding is negligible, the command to draw the checkerboard pattern takes up to $64 * 8$ bytes, which is equal to 512 bytes. This accounts for only 6.25% of the raw frame size. Note that the length of the command increases with the complexity of the drawn pattern, but it generally requires fewer bytes than transmitting a raw video or streaming a compressed video in applications where the

drawn image is relatively simple. For example, the display could be integrated into the ambient lighting system described in Section 3.2.1 to display basic navigation commands, such as turning left/right arrows or a stop signal. Additionally, using compression introduces the need for a more advanced LED module capable of decompressing the video stream, adding complexity to the system.

In contrast, transmitting a raw frame, as described in the remote control system, requires sending at least 8 KB per frame and the overhead added by RCP and its transport mechanism. Since 8 KB exceeds the typical minimum transfer unit of Ethernet, each frame must be segmented into multiple messages. An 8 KB frame, including the RCP header, needs 6 to 7 Ethernet frames when transmitted with an MTU of 1500. When going through the network, these frames will experience different queuing delays or take alternate paths, which leads to an out-of-order arrival at the receiver. A reordering mechanism at the RCP or transport protocol level is required to construct the complete frame before processing. The RCP stream must ensure the reliable delivery of messages by guaranteeing the availability of the required bandwidth and properly handling frame loss. In our use case, the loss of a single packet results in the loss of an entire image. Furthermore, although the in-vehicle network should provide sufficient bandwidth to accommodate traffic from multiple interfaces, this approach introduces more latency and jitter than the traditional implementation. Essentially, RCP must rely on a lightweight and reliable transport mechanism to ensure message delivery to the RCP nodes within the firm system requirements.

3.3 Requirements

In this section, we highlight the high-level requirements for RCP and examine the requirements and characteristics of the protocol's transport mechanism.

3.3.1 Interaction model

RCP should support different interaction models to cater to the diverse requirements of its use cases. Figure 3.11 presents an overview of the different communication patterns in automotive networks. We make the distinction between service-oriented communication patterns and traditional signal-oriented patterns. Service-oriented communication is driven by the client. Transactions are only initiated by the client when needed. Client/Server interaction is dynamically established at startup or runtime. The client relies on service discovery mechanisms to locate the appropriate service and establish communication with one of its endpoints. In contrast,

data in signal-oriented communication is published, disregarding the presence of a subscriber. The communication is configured at design time, with all identifiers and communication channels statically defined [41]. RCP does not align properly with both concepts. RCP operations

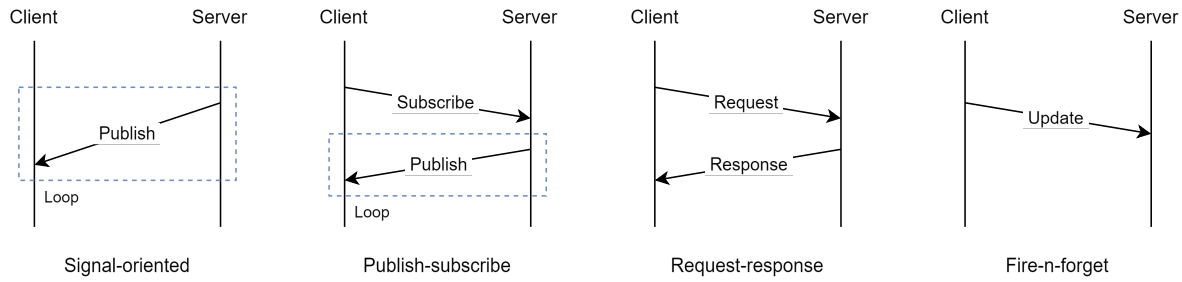


Fig. 3.11: Communication patterns in automotive networks (adapted from [41])

are mainly initiated by the client (controller or subscriber); however, RCP transports low-level operations, which misaligns with the definition of a service-oriented architecture. In SOA, services provide an abstraction of low-level functions, enabling the interaction with these functions through a higher-level interface. Modern EEAs consists of both service-oriented and signal-oriented communication associated with a signal-to-service communication mechanism. RCP exists at an intermediate point within the signal-to-service communication but cannot be described as a signal-based communication. Additionally, the operation and configuration of the RCP endpoints could be seen as service-based communication where the RCP node provides access to the nodes and their underlying functionality through an abstraction layer. In essence, RCP could be considered a bridge, combining low-level operations with service-like functionality.

By considering the use cases of Section 3.2, it is evident that RCP must rely on all the service-oriented communication mechanisms described in Figure 3.11:

1. R1 Request-response : for acknowledged transactions, such as I2C and SPI read and write operations
2. R2 Fire-and-forget: for streaming use-cases such as the PWM output control
3. R3 publish-subscribe: for notification-based use cases, such as handling interrupts on a GPIO pin. The controller subscribes to the interrupt service; then, the publisher sends an unsolicited message either once or every time an interrupt is triggered.

3.3.2 Resource usage

The implementation of RCP is driven by the software centralization trend and the need to simplify edge node complexity, ultimately reducing their production cost. As a result, RCP nodes will have limited resources. They are expected to run minimal software with a limited flash and internal memory to buffer operations and handle the decoding and decoding of RCP frames. The node must maintain little to no state. Additionally, RCP should be simple enough to enable hardware or hardware-accelerated implementations R4 .

The choice is between implementing RCP nodes entirely in hardware or utilizing a basic MCU with a minimal software stack. A hardware-based implementation offers higher performance, but it introduces challenges when implementing complex features such as service discovery. In contrast, an MCU-based implementation provides more design flexibility but comes at the cost of increased latency, overhead, and price. The MCU-based approach may also allow for limited software updates but is more likely to maintain a fixed software version throughout the node's lifetime. There exists a tradeoff between the two approaches. The choice depends on the system requirements. A hybrid approach that combines MCU with hardware accelerators might offer a balanced solution. Additionally, RCP could define complex features such as service discovery as an optional layer on top of the regular RCP operation. Therefore, this allows the implementation of statically configured hardware-based nodes and more feature-rich MCU-based nodes. The decision of what node to use depends on the manufacturer's cost analysis and the type of the controlled device.

3.3.3 Timing requirements

Automotive control signals need to meet stringent timing requirements. Relying on a remote control system implies moving the control loops to the central computing unit or the zonal controllers. This comes at the cost of increased latency and jitter, which significantly deteriorates the quality of the control signals. Therefore, it is a must to optimize the end-to-end latency of the system. The latency is mainly affected by the in-vehicle network, transport protocol, and the overhead from the zonal controller and RCP node implementation. Additionally, the system must ensure proper coordination between its various components to maintain consistency and determinism. Applications should be able to interface actuators and sensors across different vehicle zones in a well-defined order of operations. Thus, RCP should add a minimal latency overhead R5 and must inherently support features synchronization between its endpoints

R6 .

3.3.4 Discovery

Traditionally, communication links within the vehicles are statically configured. In the context of RCP, this means that the address and identifiers of the RCP endpoint are configured during design time. Automotive production lines are subject to significant variability. A peripheral or RCP node and its peripheral combination may differ slightly from one facility to another. As a result, the application is required to handle these variations, adding more complexity to its implementation since its configuration must match the actual deployment in the system. These could be avoided by implementing a discovery mechanism that detects the deployment of RCP nodes and their peripherals at runtime. It would also mitigate the risk of misconfiguration. Consider an example on which an RCP application interfaces with an I2C sensor. In a static configuration, the application is pre-configured with RCP node's ID and the specific interface the sensor is connected to. If the sensor is moved to another interface or even to another RCP node with a different address, the application needs to adapt to that change. By implementing a discovery mechanism, it is possible to assign a global identifier to the sensor, which remains statically configured at the edge node. When the application starts up, it relies on the discovery process to locate and initiate communication with the sensor. However, it is important to note that a discovery mechanism comes at the cost of increased resource usage, thus more complex and expensive edge nodes. It is preferable that RCP supports service discovery while satisfying other performance and implementation requirements R7 .

3.3.5 Operation and transport protocol

In this section we will highlight some of the low-level requirements of the protocol's operation and its relation to the transport mechanism.

Operations and Acess control

Identification R8 An RCP message is sent to a specific RCP endpoint. The routing of the message requires the endpoint to process a unique identifier. However, this ID could possibly be unique only locally to the node it belongs to. This is possible because part of the identification could also be delegated to the node itself. With the protocol transported over Ethernet, each network node will be identified by its MAC address, which could be combined with the local endpoint ID to identify it uniquely. The same applies if a higher-level transport protocol (e.g.,

UDP) is considered, except that IP addresses are used instead of MAC addresses. In general, the choice of endpoint IDs in RCP will depend on the system design. There are two options: assigning a unique system ID to each node or offloading part of the ID responsibility to the RCP node, thus depending on the transport mechanism.

Multicast and Broadcast operations R9 Multicast operations could occur at various levels: The controller may directly address multiple RCP nodes or send the same operation to multiple endpoints on the same or different RCP nodes. In the latter case, the protocol itself must handle multicast operations. However, in the former scenario, the transport protocol needs to support multicasting/broadcasting.

Compound and atomic operations R10 An atomic operation refers to an uninterruptible action that is either executed or is not executed at all. For example, the $transmit\_receive$ operation is atomic because it can not be split into two separate $transmit$ and $receive$ operations. The $received$ data depends on the command/address in the last $transmit$ operation. If another transmission occurs in between, the operation outcome is incomplete. Atomic operations are not limited to one interface; multiple operations on different interfaces could share their atomicity relationship. Consider 2 GPIO pins on separate interfaces that should remain consistent. These pins are updated using a compound RCP operation. The system must ensure that all operations are executed successfully. The failure of one GPIO operation means the failure of the whole operation. Handling a failed operation depends on the type of peripheral. For a GPIO operation, the RCP node can revert any changes made to the output of a pin; however, for bus operations, failures can not be reverted as the RCP node lacks the context of the overall communication state. Thus, it is only required to notify the controller of the operation failure and allow it to handle the error. In essence, RCP should support the grouping of operations into a compound operation, which can either be atomic or not.

Scheduling and periodic operations R11 RCP could define a mechanism to perform periodic operations locally to reduce the overall network traffic. For example, if an application is polling a temperature sensor every 1 ms, it needs to send the same $transmit_receive$ operation over and over again. Instead, the RCP node could automatically replicate the request and send a response to the controller, eliminating the need for redundant messages over the network. It could implement basic operations locally, such as performing the same operation every 't' seconds for 'n' times. Operations could also be bound by conditions, such as performing the

same operation every 't' seconds until a GPIO input changes.

Additionally, RCP could define a mechanism to schedule operations to be performed at a specific point in time. This is helpful in two scenarios: first, scheduling operations allows for the flexibility to schedule multiple operations to be executed concurrently at the same time. Two, the jitter could be reduced by slightly delaying the execution of the operation to account for the discrepancy in transmission times.

In the next section, we provide a short overview of the requirements from the perspective of transport protocol.

Message handling

Message mapping R12 The transport protocol, or RCP, should map response messages to their corresponding requests. This ensures multiple requests/responses are sent, received, and processed, irrespective of their order.

Aggregation R13 RCP should support aggregation of multiple operations, or RCP messages, into a single RCP frame. This allows the implementation of compound and atomic operations and reduces the protocol's network overhead.

Communication and synchronization

Transport layer R14 Ideally, RCP frames should be transported over any network layer: Layer 2 (Ethernet), Layer 3 (IP), or Layer 4 (TCP/UDP); however, the protocol should at least support Layer 2 transport for it to operate over an Ethernet network.

Encoding R15 The transport protocol for RCP operations must use a suitable encoding for all of its operations, with the option to use a fixed or variable length encoding.

Timestamping R16 The transport protocol for RCP should include timestamps in its frames to ensure that it is compatible with synchronization mechanisms, such as gPTP, for scheduling and consistency of RCP operations.

Traffic handling R17 The protocol should be compatible with traffic shaping and policing mechanisms for time-sensitive networks.

Error handling R18 While the protocol should implement error handling in cases of failure of an RCP operation, the transport mechanisms should handle errors related to communication. This includes the detection of bit errors, packet loss, connection loss, and packet reordering. The transport mechanism should ensure reliable transport of RCP messages.

Segmentation R18 Used in case of unreliable communication to transport messages that exceed Ethernet's MTU.

Additional requirements

We establish the following extra requirements that are important for the transport protocol from an implementation and usage perspective:

- Network overhead R20
- SW availability R21
- Compute load R22
- Startup time R23

3.4 Standardization

RCP is currently under standardization by Open Alliance, a consortium of leading industry companies that aim to promote the adoption of Ethernet-based networks as the standard for automotive applications. RCP is currently defined under the TC18 committee of Open Alliance. The standardization focuses on meeting the industry's requirements for RCP, including designing the overall system on which the protocol is deployed and defining a specification for the protocol messages and interaction model [42].

4 RCP implementation using existing communication protocols

In this section we provide a detailed implementation of the two use-cases discussed in Section 3.2: The PWM/Motor driver control and the OLED display control scenarios. We perform qualitative and quantitative analysis of IEEE1722 and SOME/IP and their suitablity to be used as a transport mechanism for a remote control protocol.

4.1 Measurement Methodology

All of our evaluations were done using off-the-shelf components. This includes devices such as the Raspberrypi 4B [43] and the Raspberrypi 5[44], in addition to SPI and I2C devices. We measured the following parameters:

- **Latency :** refers to the time between two points in our program. It could represent the time between the initiation of an operation and its arrival at its destination. For instance, the Latency between point A, when a request is sent from the Talker application, and point B when the Listener application completes processing the request.
- **RTT (round-trip time):** refers to the total time it takes to execute an operation, including sending a request and receiving a response. It only applies to request-response operations.
- **CPU usage %:** refers to the percentage of the total CPU capacity used by an application to perform a single operation. It could be measured between two points in the program, allowing us to measure the CPU usage for certain implementation components.
- **Rate:** refers to the frequency at which a certain event happens. It is measured by calculating the period between consecutive occurrences of the event over a period of time.

4.1.1 Timing measurements

In most of our evaluated scenarios, the Talker and Listener applications are executed on different nodes on the network (e.g., the Talker is running on the vehicle computer, and the Listener is running on the edge node). To achieve correct measurements, we ensured that all nodes were synchronized to the same time. This allowed us to measure Latency, or round-trip time, by

inserting timing probes that record the current system time. The collected time points are then post-processed to extract the timing data. Since all our devices are based on Linux, we opted for LinuxPTP [45], an open-source software implementation of the generalized Precision Time Protocol for synchronization. gPTP works by selecting one node as the GM clock to provide timing information to all other nodes in the network. The selection is made using the Best Master Clock Algorithm (BMCA), which establishes the best clock in the network based on metrics such as clock quality, accuracy, and the stability of the local oscillator, in addition to user-set parameters such as the clock priority. Synchronization is achieved by measuring the offset between the GM clock and other devices on the network using timestamps from the gPTP message exchange. The system clock is then adjusted to match the GM clock [46]. Timestamps are either generated in software or hardware using specialized NICs (network interface cards). Hardware timestamping generally provides better synchronization as it eliminates the jitter introduced by the network stack. However, we opted for software timestamping in our setup because not all our NICs support it. As presented in Figure 4.1, all of our devices are connected to the same 100BASE-TX Ethernet network through a layer 2 switch. We configured a Linux service to start LinuxPTP over the Layer 2 network. Synchronization is done using the ptp4l program included in the Linuxptp package. When operating in software timestamping mode, ptp4l directly synchronizes the system clock to the GM clock [47].

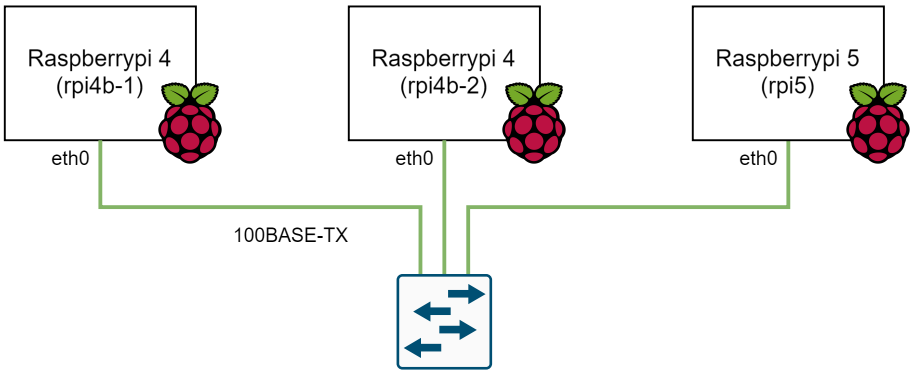
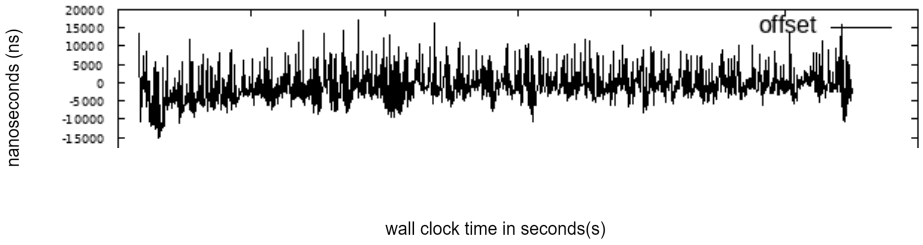
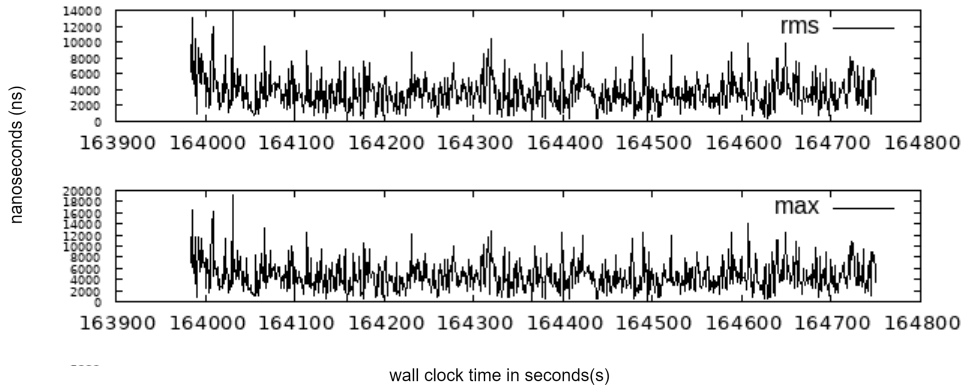


Fig. 4.1: Ethernet network for time synchronization

We experimented with different configuration settings for ptp4l. The application provides multiple algorithms for synchronization, which are set by the $clock\_servo$ option. We experimented with the default pi option and the adaptive $linreg$ option. We ran each option for around $15 - 20$ minutes and recorded their clocks. Figure 4.2a shows the system clock's offset of one of the nodes to the GM clock when using the pi servo, while Figure 4.2a shows the rms and



(a) pi servo



(b) $linreg$ servo

Fig. 4.2: System clock offset to the GM clock

maximum values of the clock offset when using the $linreg$ servo. In both cases, the clocks are synchronized to $\pm 15 \mu s$, which is deemed enough for our measurements, so we decided to use the default pi option. A timepoint is recorded by reading the current value of the system's CLOCK_REALTIME, which represents the system-wide clock [48] and is adjusted to match the GM clock. The application reads the clock in seconds and nanoseconds and then logs the timestamp.

4.1.2 Additional measurement tools

To measure the load between two points in the program, we log the number of jiffies used by both the CPU and the application during this interval. A jiffy represents the basic time unit in the Kernel, which is determined by the Kernel's tick rate [49]. We can determine the processing time of an application by tracking the number of jiffies it uses. CPU usage percentage is calculated using the following equation:

$$\text{CPU usage (\%)} = \left(\frac{\text{Jiffies used by the application}}{\text{Total jiffies used by the CPU}} \right) \times 100 \tag{4.1}$$

4.1.3 CPU usage measurement

We used additional measurement tools to help debug our applications and gather additional information, such as capturing the network packets. This includes the following tools:

- **PicoScope 5000 Series:** A USB oscilloscope allowing high-resolution capture for analog and digital signals [50].
- **ProfiShark 1G+:** an Ethernet TAP, designed to accurately capture network traffic without altering it and affecting its performance [51].

4.2 Head-to-head Comparison

For a head-to-head comparison between the two protocols, we implemented the PWM/Motor control scenario presented in use-cases 3 of Section 3.2.3.

4.2.1 Evaluation setup

Figure 4.3 shows our evaluation setup. We drive the LED using an Adafruit PCA9685 module [52], a 16-channel 12-bit PWM/Servo driver with an I2C interface, connected to two Raspberry PI devices which are linked via a point-to-point Ethernet connection. We developed two applications: A Talker application that contains the LED control logic and sends I2C control commands and a Listener application, which receives and executes the commands locally.

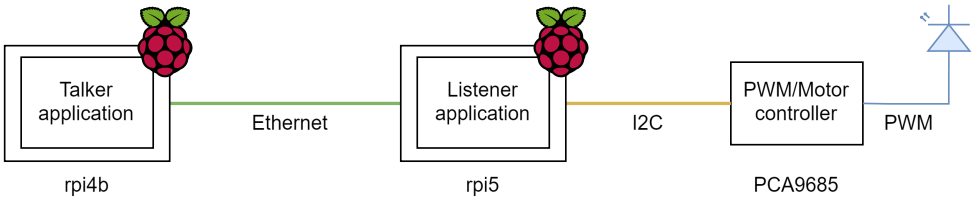


Fig. 4.3: Head-to-head evaluation setup

4.2.2 IEEE1722 Talker/Listener Implementation

We used the open-source implementation of the IEEE1722 protocol - Open1722 [2]. Figure 4.4 describes the execution flow of the Talker and Listener applications. To perform I2C operations, e.g., transmit operation, the Talker creates an AVTP frame that encodes the I2C parameters and

[2] https://github.com/COVESA/Open1722

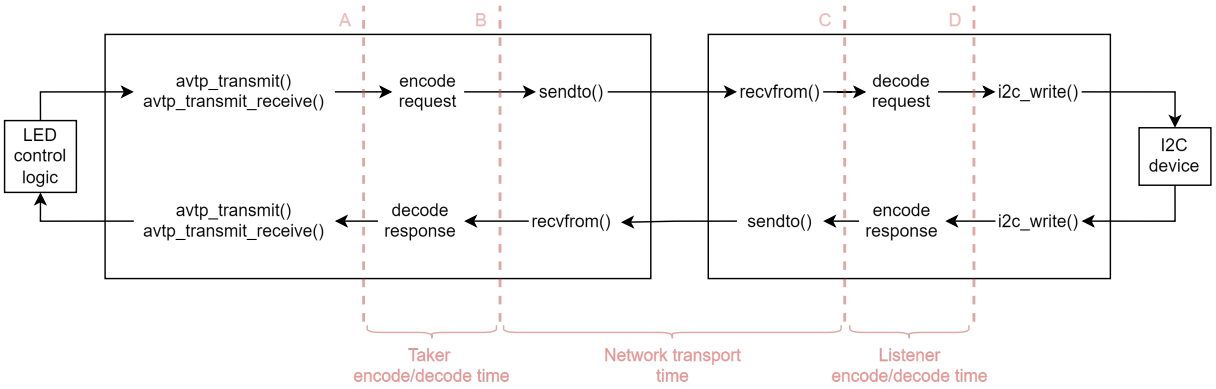


Fig. 4.4: IEEE1722 I2C Talker and Listener applications execution flow

payload. This frame is sent over the network using Linux sockets. When received by the Listener, the frame is decoded, and the I2C operation is executed locally. If a response is expected, the Talker blocks while waiting for the Listener's response. We used Open1722 to encode and decode the AVTP frames. The I2C operations are mapped to GBB frames. We defined two operations, $transmit$ and $transmit\_receive$. In the $transmit$ operation, the Talker sends a GBB request containing the I2C payload, and the Listener responds with an empty GBB message for acknowledgment. In the $transmit\_receive$ operation, the Talker sends a GBB request with the I2C payload, and the Listener replies with a GBB message containing the received I2C payload. To perform an I2C receive operation, the $transmit\_receive$ operation is used with no payload from the Talker. We mapped the I2C fields to a GBB frame as follows :

- **Target I2C bus:** In Linux I2C, are automatically assigned numbers by the kernel. We used the bus number to map the request to the correct I2C peripheral. The I2C bus number is stored in the Byte Bus ID field.
- **Operation type:** mapped to the op field. A ($'1'$) indicates a $transmit$ operation and a ($'0'$) indicates a $transmit\_receive$ operation.
- **Message type:** used the rsp field to specify whether the message is a request ($'0'$) or a response ($'1'$).
- **Status:** In case of a response, we use the err field to indicate if the operation failed. The field is cleared if the operation is successful.
- **Response:** IEEE1722 does not differentiate between fire-and-forgot and request-response communication, so we used the least significant bit of the evt field to indicate whether a

response is required ('$xx1$') or not ('$xx0$').

- **Address length:** used the 2^{nd} least significant bit of the evt field to specify the target address length ('$x0x$' for 7-bit addresses and '$x1x$' for 10-bit addresses).
- **Target address:** mapped the target address to the first two bytes of the payload.
- **Read size:** stored the number of Bytes to read in the Read size field. This only applies to $transmit\_receive$ operations; otherwise, this field is not used.
- **I2C payload:** Stored in the remaining part of the payload.

4.2.3 SOME/IP Talker and Listener Implementation

For the SOME/IP, we used the open-source implementation vsomeip [3]. The SOME/IP Talker and Listener applications work the same way as the IEEE1722 Talker and Listeners. The main difference is that the Listener is implemented as a service that exposes the I2C bus, and the Talker uses service discovery to establish communication with the Listener. Our latency measurements are taken after the service discovery process finishes. In our implementation, we mapped the (service ID, instance ID) pair to the corresponding I2C bus number (e.g., (1,0) refers to the first instance of I2C bus one service). The Service ID was always set to 0, as each bus had one dedicated service. Figure 4.5 shows the execution flow of the Talker and Listener applications. The vsomeip implementation is multi-threaded. The Talker sends a request and then blocks while waiting for a response. The $on\_repsonse()$ callback is triggered when a response is received. Similarly, on the Listener side, the $on\_transmit$ and $on\_transmit\_receive$ callback are triggered when a request is received.

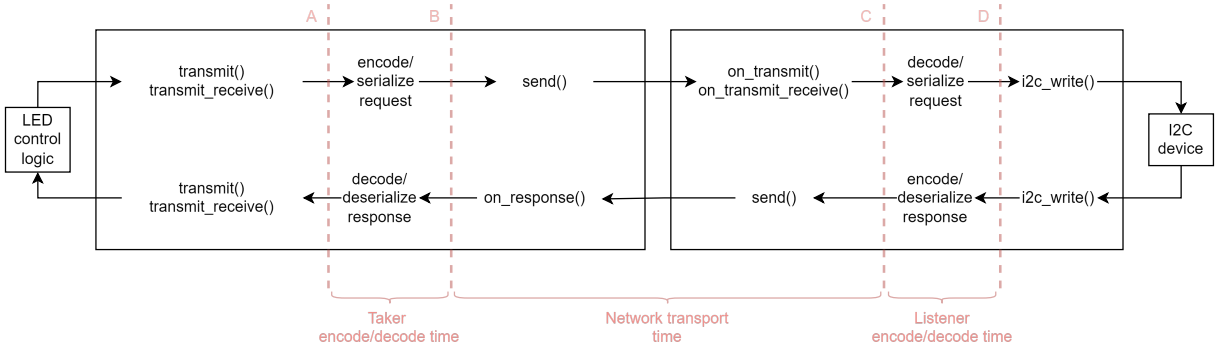


Fig. 4.5: SOME/IP I2C Talker and Listener applications execution flow

[3] https://github.com/COVESA/vsomeip

In contrast to IEEE1722, SOME/IP does not provide formats for encapsulating control signals such as I2C. Therefore, we described the operations using custom formatting. The I2C parameters are directly stored in the payload as an object:

```
{
Target address,
Read/Write size,
I2C payload
}
```

SOME/IP serialization allows the use of variable-length fields. In this case, the target address is either represented by a one-byte or two-byte field, depending on the address length used. To put everything together, We mapped the operation to a SOME/IP message as follows:

- **Service ID:** Mapped to the I2C bus number
- **Method ID:** Mapped to the operation type. We defined the $transmit$ and $transmit\_receive$ operations as methods; each method is associated with a unique method ID ($transmit$ method ID = 1, and $transmit_receive$ method ID = 2).
- **Message Type:** Indicates whether a response is required or not (can be set to REQUEST or REQUEST_NO_RETURN).
- **Return Code:** Indicates whether an operation is succesful. SOME/IP specifies a set of predefined status and error codes. We used the E_OK code to indicate a successful operation and the E_NOT_OK code to indicate failure.
- **Message payload:** Include the I2C operation parameters and payload.

4.2.4 Evaluation scenarios

We defined two scenarios:

- **Scenario 1:** The Talker sends an update every 1 ms and waits for a response from the Listener (request-response communication). The next update is delayed if a response is not received within the specified period.
- **Scenario 2:** The Talker sends an update every 1 ms but does not expect a response from the Listener (fire-and-forget operations).

Both scenarios were evaluated over a 100BASE-TX Ethernet link and a 10BASE-T1S link. For the IEEE1722 implementation, we tested the implementation over both Layer2 and UDP, while the SOME/IP implementation was tested with UDP and TCP. In the next section, we describe the implementation details of the Talker and Listener application using both IEEE1722 and SOME/IP as transport protocols for RCP. We discuss how I2C operations are mapped to RCP message. It is important to note that our focus is only on the transmit/receive operations, excluding the bus initialization process, which is directly managed by the Listener in this implementation.

4.2.5 Evaluation results

This section presents our evaluation results, which include Latency, RTT, and overhead measurements for both scenarios discussed in the previous section.

vsomeip version comparison

Initially, we selected the latest version at the time of vsomeip, version 3.5.1. However, during our initial measurements, we noticed that the vsomeip-based implementation is significantly slower than the IEEE1722 implementation [4]. In addition to the achieved RTT not aligning with previous evaluations of vsomeip [53]. To further investigate the issue, we used the same benchmarking utility in [53] to replicate RTT measurement for different versions of the library [5]. We noticed that RTT increased with the major update from version 2 to version 3. Running the benchmark with version 2.10.11 over a 100BASE-TX connection produced an RTT value ranging from 300 to 400 μs; however, when using version 3.5.1, this range increased to around 12,000-13,000 μs. The reason behind this discrepancy in performance is unknown and is out of the scope of this work, so we opted for using version 2.10.11 for the remainder of our evaluations, which we will refer to as vsomeip2.

Scenario 1

Figure 4.6 presents the round-trip time and load measurements for the open1722 and vsomeip implementations of scenario 1. The measurements were collected from $10,000$ I2C $transmit$ operations. RTT was measured as the average time between a $transmit$ operation is sent until

[4] https://github.com/COVESA/vsomeip/discussions/785

[5] https://github.com/kamelfakihh/vsomeip-docker-benchmark

its response is received. Therefore, RTT represented the average execution time of the whole operation, which was measured using probe 'A' shown in figures 4.4 and 4.5.

For the open1722 implementation, our evaluations show a slight increase in the round-trip time when encapsulating the AVTP frame in a UDP packet compared to an Ethernet packet. The results present approximately a 6% increase in RTT when sending the packets over both 1000BASE-TX and 10BASE-T1S Ethernet. However, the partial overlap in the error bars indicates that the difference is insignificant. The same could be said about the increase in RTT observed in the vsomeip2 implementation between UDP and TCP encapsulation.

Comparing the open1722 and vsomeip2 implementations, we observed that the open1722 performed better over both Ethernet links, with approximately 20% improvement in performance. The open1722 and someip2 implementations met the 1ms deadline when transported over 100BASE-TX except for the SOME/IP TCP encapsulation. On the other hand, both applications failed to meet the deadline when relying on 10BASE-T1S, which runs at a slower speed.

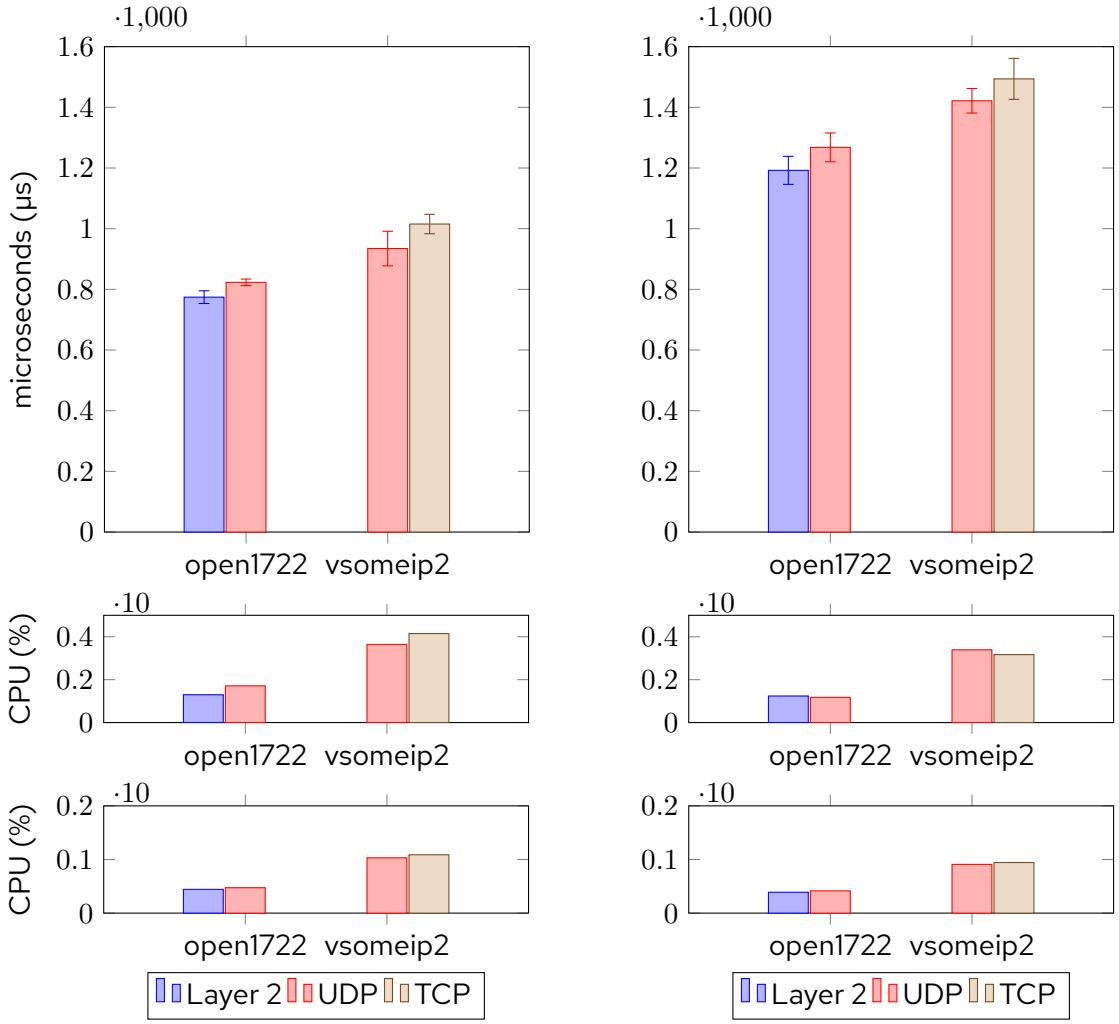
To further investigate the factors contributing to the overall round-trip times, we measured the contribution of the following operations to the RTT:

- **Encoding/Decoding time:** The time required to serialize and deserialize the request and response at both the Talker and Listener. It was measured between probes 'A'-'B' and 'C'-'D' shown in figures 4.4 and 4.5.
- **Transmission time:** The time required to send the request and response frames, which includes the time the frame is processed by the protocol stack, the network stack, and the on-wire transmission time. It was measured between probes 'B' and 'C'.
- **I2C operation time:** The time required to write a command to the I2C bus.

Figure 4.7 shows the contribution of each of these values to the 10BASE-T1S evaluation. We observe that both protocols' encoding/decoding operations are in the same ballpark. The main performance difference between the two implementations is caused by the protocol's transmission time. Finally, we can observe that the vsomeip2 implementation consumes more than double the CPU usage of the open1722 implementation. This could be observed for both the Talker application (2nd row) and Listener (3rd row) applications in Figure 4.6.

Scenario 2

Figure 4.8 presents the latency and load measurements for the open1722 and vsomeip implementations of scenario 2. The measurements were also collected from $10,000$ I2C $transmit$



(a) 100BASE-TX

(b) 10BASE-T1S

Fig. 4.6: Head-to-head RTT and CPU usage % for scenario 1 over 100BASE-TX and 10BASE-T1S.

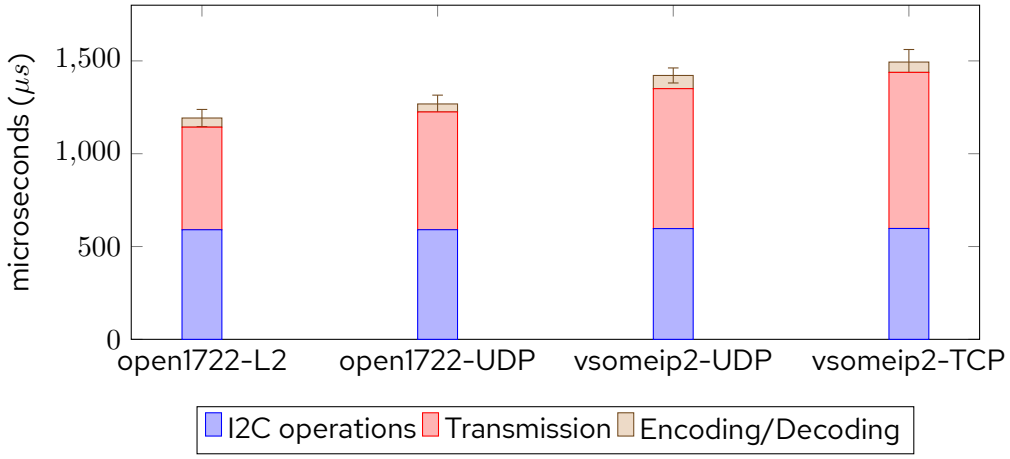
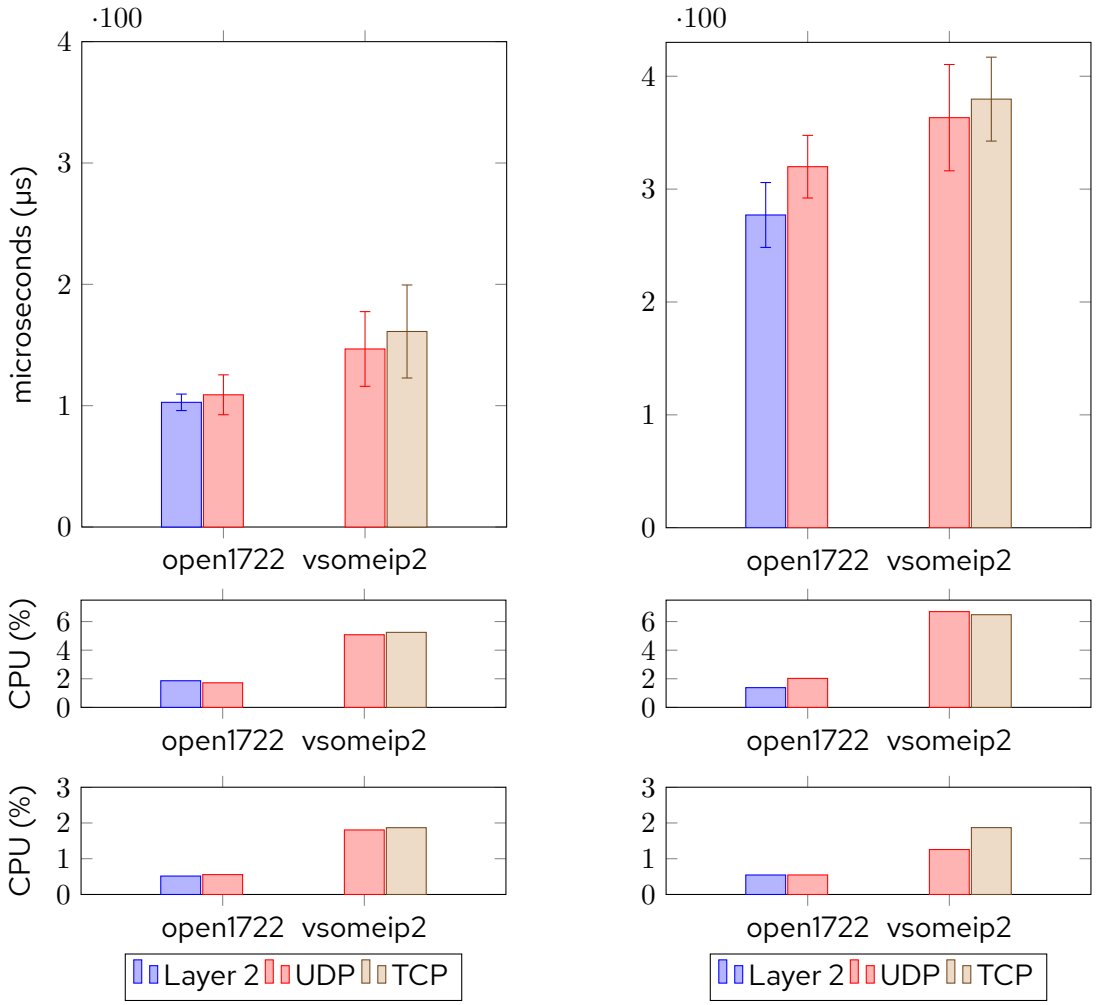


Fig. 4.7: In depth Head-to-head RTT comparison between scenario 1 implementations over 10BASE-T1S

operations. We measured the Latency as the duration between the moment a $transmit$ operation is started and the time the I2C message starts to be written to the bus. It is measured between probes 'A' and 'D'. Similarly, we can observe a slight increase in Latency from L2 to UDP encapsulated AVTP frames and from UDP to TCP encapsulated SOME/IP message. However, we observed a more significant performance difference between vsomeip and open1722 applications, where the open1722 implementation is approximately $30 - 40\%$ faster. This percentage increase is expected as we did not account for the I2C operation in the Latency measurements. The goal is to measure the protocol's Latency without the influence of the I2C operation.



(a) 100BASE-TX

(b) 10BASE-T1S

Fig. 4.8: Head-to-head latency and CPU usage % measurements for scenario 2 over 100BASE-TX and 10BASE-T1S.

4.3 End-to-End Performance

For an end-to-end comparison between the two protocols, we implemented the OLED control scenario presented in use-case 4 of Section 3.2. We used off-the-shelf components to create a make-shift zonal architecture. Then, we evaluated the end-to-end performance of both IEEE1722 and SOME/IP as transport mechanisms for RCP.

4.3.1 Evaluation setup

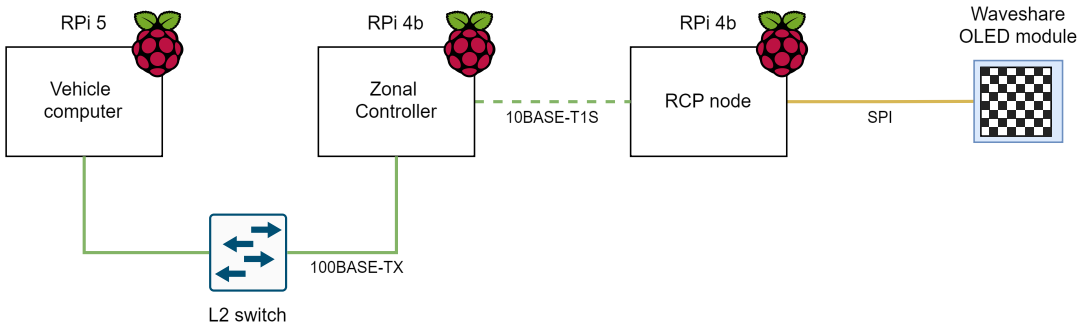


Fig. 4.9: End-to-end evaluation setup

For the display, we selected a 128x128 Waveshare OLED display [54]. The display provides a 4-wire SPI interface with the exact specifications discussed in Section 3.2.4. The vehicle computer is represented by a Raspberry PI 5, and both the zonal controller and the RCP (edge) node are represented by two Raspberry PI 4s. The zonal controller is linked to the vehicle computer by a 100BASE-TX Ethernet connection, while the RCP node is linked to the zonal controllers via a 10BASE-T1S connection. The 10BASE-T1S bus consists of only two nodes and uses two USB-to-10BASE-T1S adapters from Microchip (EVB-LAN8670-USB [55]). The components used are not of automotive grade but are close enough to what is expected in an automotive network. The processing power of the Raspberry PI 5 is comparable to the processing power of a vehicle computer. However, the Raspberry Pi 4 used for the controllers and edge nodes is more powerful than their typical automotive equivalent. The selection of these nodes was limited by the available software implementation for SOME/IP, which is POSIX compatible only so that we couldn't use non-POSIX devices (e.g., a microcontroller).

4.3.2 Implementation

We implemented the following components:

- **OLED controller:** The application is responsible for issuing drawing commands. (e.g., drawing a rectangle with width w and height h at coordinates (x,y)). It can either directly interface a local instance of the Renderer or use the Renderer client/service to communicate with a remote instance of the Renderer.
- **Renderer:** A library to process the drawing commands. It receives a list of commands and renders the corresponding image, resulting in a raw 128x128 image output. The image is rendered using the lightweight embedded graphics library, LVGL [6].
- **Renderer service/client:** SOME/IP client/service pair used to establish communication between the OLED controller and the Renderer. The drawing commands are sent using a custom encoding that describes each shape to be drawn. Therefore, drawing functionality is provided as a service that abstracts the rendering to the display. We refer to this service as the Drawing service.
- **Display driver:** A custom driver implementation to operate the display. It can work with a locally connected display, or use IEEE1722 and SOME/IP Talkers/Listeners to remotely operate the display.
- **SPI Talker/Listener:** Used by the driver to send RCP messages to the edge node connected to the display.

The SPI Talker and Listener applications were implemented in a similar fashion to the I2C implementation of Section 4.2.3. For this use case, it was only required to implement the SPI write operation. An image is drawn to the display by sending a command containing its starting coordinates (row, column), followed by a byte stream containing the actual image data. To achieve this, we implemented a $transmit$ operation that includes both the command and data message payload. For the IEEE1722 Talker/Listener, the SPI write command is similarly mapped to a GBB frame, and the fields are used in the same manner. There are three differences:

- **evt:** All of the display operations are using a fire-and-forget mechanism, so this field is changed to define the command length, such that when the Listener receives the message, it can split the command and data payloads and locally handle the DC line of the SPI peripheral.
- **Byte Bus ID:** Both the SPI bus number and the CS line number are mapped to the Byte Bus ID, such that the LSB half of the field represents the chip select number, and the MSB

[6] https://lvgl.io/

half represents the SPI bus number. The chip select signal is also handled locally by the Listener.

- **Payload:** The command and data payload are stored directly in the GBB payload.

As for the SOME/IP Talker and Listener, the vsomeip version we are using does not support SOME/IP-TP, meaning that it is not possible to use segmentation, which is required in this scenario as the raw frame is up to 8 KB. For this reason, we used SOME/IP to encapsulate and transport GBB messages, relying on Open1722's encoding and segmentation mechanism without the need to define a new encoding for SPI operations. We defined an $avtp_request$ method that tunnels the IEEE1722 operations.

Finally, we defined a custom encoding for each drawable that can be drawn on the display. A drawable could be a label, rectangle, circle, or a generic acute polygon. These encodings are based on the LVGL properties of the drawables. The drawables are added to a list, which is sent from the OLED controller to the Renderer. The Renderer then sequentially decodes the properties of each drawable and puts it into the image. For example, A rectangle drawable is encoded using the following object:

```
{ /* Dimensions */
uint8_t x,
uint8_t y,
uint8_t w,
uint8_t h,

/* Border */
uint8_t BorderColor,
int16_t BorderWidth,
uint8_t BorderOpacity,

/* Shadow */
uint8_t ShadowColor,
int16_t ShadowWidth,
int16_t ShadowOffsetX,
int16_t ShadowOffsetY,
int16_t ShadowSpread,
uint8_t ShadowOpacity };
```

The Renderer implementation does not support LVGL animations, so the complete image encoding is updated with every frame. The image drawn in our evaluation consisted of 32 squares arranged into a checkerboard pattern, which is moved slowly across the screen with every frame.

4.3.3 Evaluation scenarios

We evaluated the performance in four different scenarios:

- **Scenario I:** The baseline scenario involves the OLED controller and Renderer applications directly running on the edge node without relying on the display service. In this scenario, the drawing operation has the least overhead.

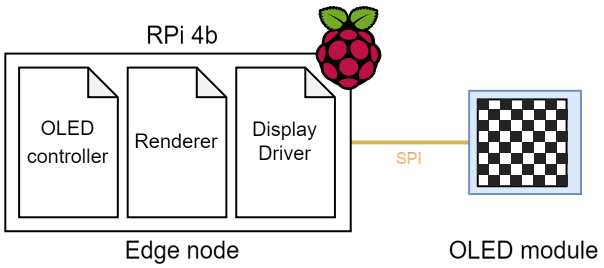


Fig. 4.10: End-to-end comparison scenario I

- **Scenario II:** In the traditional scenario, the Rendering service runs on the edge node, while the OLED control application runs on the vehicle computer. The controller relies on the Renderer client/service to send drawing commands to the Renderer.

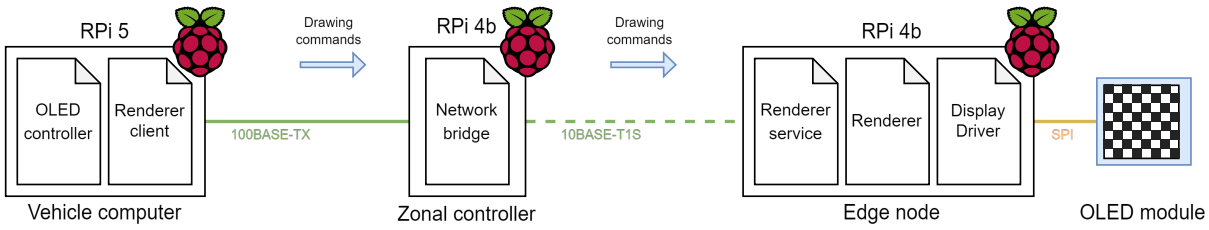


Fig. 4.11: End-to-end comparison scenario II

- **Scenario III:** Using IEEE1722 as RCP. The Renderer and display driver are both moved to the vehicle computer. The raw image is now generated on the vehicle computer and sent to the Edge node using RCP.

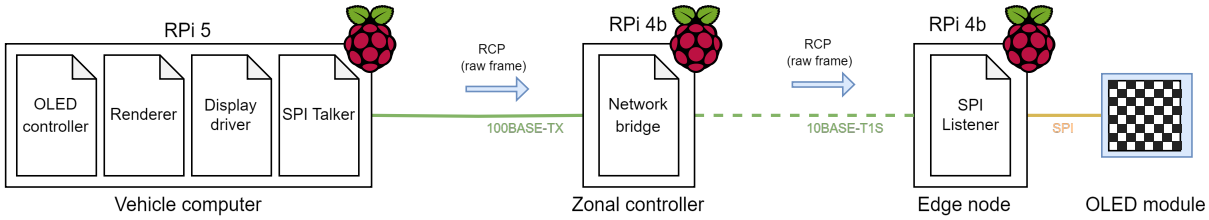


Fig. 4.12: End-to-end comparison scenario III and IV

- **Scenario IV:** Using SOME/IP as RCP. It follows the same implementation of the previous scenario. However, the Listener and Talker applications are replaced by their SOME/IP version.

It is important to emphasize that the zonal controller acts as a bridge between the 100BASE-TX and the 10BASE-T1S Ethernet segments. Ideally, the zonal controller could be replaced by a switch; however, we didn't have any replacement available, so we used a Raspberry PI and set up a bridge in software using Linux bridge-utils.

4.3.4 Evaluation Results

We measured the end-to-end Latency for each scenario in Figure 4.13. The Latency is defined as the time between the moment a drawing command is issued and the time the frame is drawn to the display. We observe that the traditional ECU setup (Scenario II) achieves a similar performance to the baseline measurement (Scenario I). Meanwhile, for the RCP scenarios III and IV, the Latency increased by 40 to 45% compared to the baseline scenario. This is attributed to the increased size of data to be transmitted over the network. Additionally, we observe that the open1722 implementation is slightly faster than the vsomeip2 version. However, the results are still comparable, indicating that the performance difference observed in our head-to-head evaluations is less evident as the data size increases. This is because messages are segmented and processed parallel to their transmission, thus reducing the protocol stack overhead.

We measured the target and achieved framerates in Figure 4.14. The target framerate is measured as the rate at which the Talker application sends drawing commands/frames, while the achieved framerate is measured as the rate at which frames are drawn to the display on the Listener's end. All scenarios successfully achieved the target average frame rate of 80 fps. However, we observed a higher frame rate jitter when using the SOME/IP implementation.

Finally, we calculate the overhead for each of our RCP implementations using the following equation:

$$Overhead = \text{Total header length} \cdot \lceil \frac{\text{Data length}}{MTU - \text{Total header length}} \rceil \tag{4.2}$$

For the IEEE1722 as RCP implementation, the total header length is:

$$\text{Total header length} = H_{\text{eth}} + H_{\text{ntscf}} + H_{\text{gbb}} = 14 + 12 + 16 = 42 \text{ Bytes} \tag{4.3}$$

While for the SOME/IP as RCP implementation, the total header length is:

$$\text{Total header length} = H_{\text{eth}} + H_{\text{ip}} + H_{\text{udp}} + H_{\text{SOME/IP}} + H_{\text{gbb}} = 14 + 20 + 8 + 16 + 16 = 74 \text{ Bytes} \tag{4.4}$$

Assuming the MTU is set to 1500 Bytes, the transmission overhead for our 8,192-byte message is 1.75 % in the open1722 implementation, compared to 5.41 % in the vsomeip2 implementation. With a bigger header, the vsomeip2 overhead becomes higher for smaller payloads.

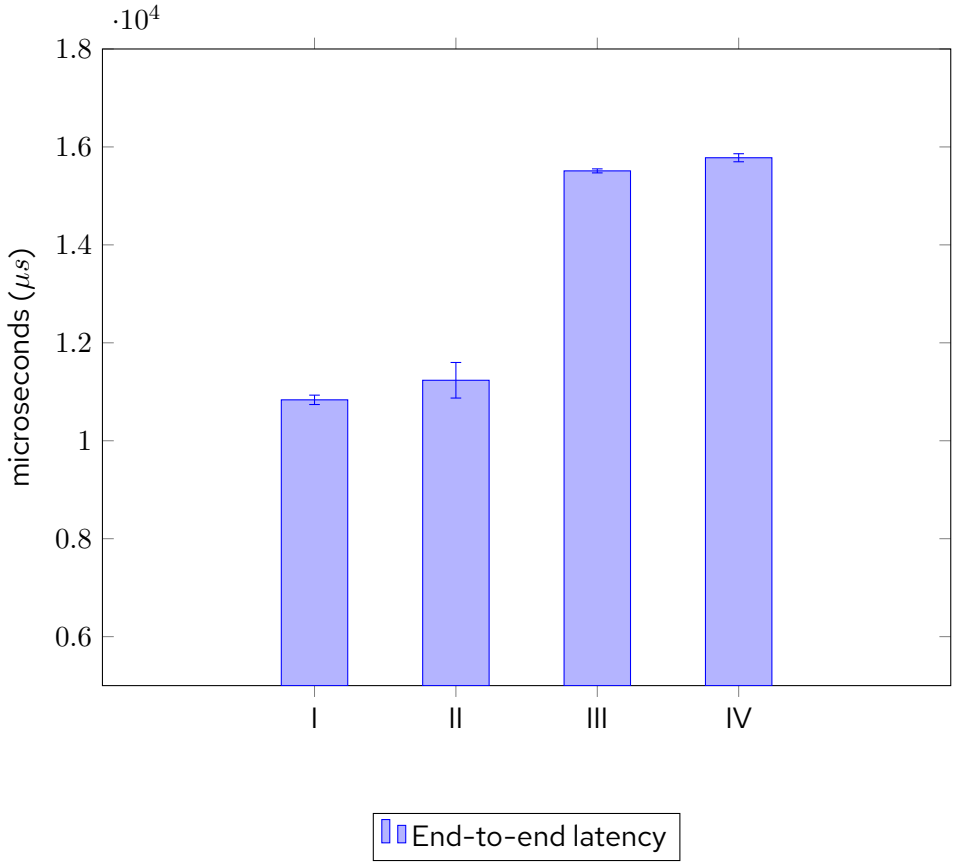


Fig. 4.13: End-to-end frame latencies for scenarios I to VI

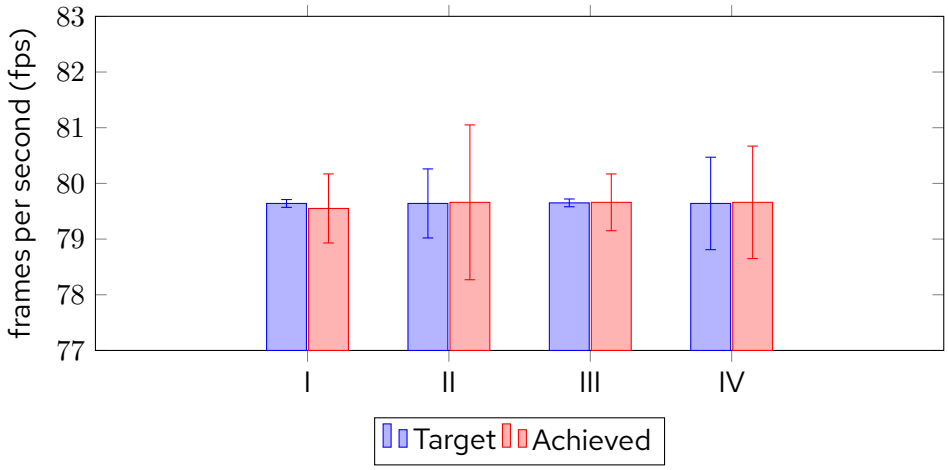


Fig. 4.14: End-to-end frame rates for scenarios I to VI

4.4 Discussion

In this chapter, we looked into implementing IEEE1722 and SOME/IP as remote control protocols for I2C and SPI peripherals. We covered the interaction model, mapping the remote peripheral's operations to protocol messages and performance analysis. Our latency and round-trip time evaluations show approximately a 50 % increase in transmission latency when using SOME/IP with UDP encapsulation over IEEE1722 with Layer 2 encapsulation. We observe that the end-to-end latency of our request-response operations was mainly bottlenecked by other factors, such as the network transmission time and the blocking delay when writing/reading to and from the interface, rather than the overhead introduced by the protocol stack. We also observed an insignificant change in performance when encapsulating the same protocol on different network layers (i.e., L2/UDP and UDP/TCP). Therefore, we conclude that while IEEE1722 offers better latency, both IEEE1722 and SOME/IP are still good candidates from a latency perspective, as long as the underlying network provides sufficient bandwidth. We also infer that relying on higher network layers imposes a minor performance penalty which allows the protocol to be transported on any layer based on the specific application requirements. For instance, TCP encapsulation can be used for reliable communication, while UDP and Layer2 encapsulation is preferred for critical applications, given that the protocols are agnostic of the transport layer. IEEE1722 offers this flexibility as it is a Layer 2 protocol with a UDP encapsulation (that can be extended to TCP encapsulation), while SOME/IP demands the use of UDP/TCP transport. We also observed that the computational load of SOME/IP applications was significantly higher than the IEEE1722 applications across all evaluated scenarios. Additionally. while we did

not implement any scheduling mechanisms, we observed higher jitters when using SOME/IP compared to IEEE1722. This could be attributed to the bulkier implementation of SOME/IP, highlighting the requirement for hardware-accelerated solutions.

Finally, we present a side-by-side comparison of SOME/IP and IEEE1722 in Table 4.1 considering the requirements outlined in Section 3.3 and the challenges we faced during the implementation.

Tab. 4.1: Side-by-side comparison of SOME/IP and IEEE1722 in the context of remote control protocols. "+"represents better performance, "presents worse performance, and -ïndicates comparable performance

Requirement	Description	IEEE1722	SOME/IP
R1	Request-response communication	~	~
R2	Fire-and-forget communication	~	~
R3	Publish-subscribe communication [7]	-	+
R4	Hardware acceleration	+	-
R5	Latency overhead	+	-
R6	Synchronization	+	-
R7	Service discovery	-	+
R8	Identification	~	~
R9	Multicast and Broadcast operations	~	~
R10	Compound and atomic operations	+	-
R11	Scheduling and periodic operations	+	-
R12	Message mapping	~	~
R13	Aggregation	+	-
R14	Transport Layer [8]	+	-
R15	Encoding	+	-
R16	Timestamping	+	-
R17	Traffic handling	+	-
R18	Error handling	~	~
R19	Segmentation	~	~
R20	Network Overhead	+	-
R21	Software availability	~	~
R22	Compute load	+	-
R23	Startup time	+	-

[7] IEEE1722 is a connectionless protocol, it does not support subscription mechanisms
[8] SOME/IP is limited to UDP/TCP, while IEEE1722 can operate on top of any network Layer

5 Concept for a new remote control protocol

In this section, we provide a concept for a remote control protocol that combines parts from SOME/IP and IEEE1722 to implement a feature-complete remote control system.

5.1 Introduction

While our evaluations show comparable performance in terms of latency and round-trip times between the SOME/IP and IEEE1722 implementations, we can infer from the comprehensive analysis of the two protocols presented in Table 4.1 that IEEE1722 is more suitable for the streaming and control applications required by RCP. It already supports control formats for key automotive buses and peripherals, and it integrates well into time-sensitive networks by supporting important features such as traffic management and synchronization. It is also lightweight and potentially easier to implement in hardware. However, these features alone fall short of satisfying the service-oriented nature of some RCP requirements. For instance, since IEEE1722 is a connectionless protocol, it does not support publish-subscribe operations. It is also statically configured and does not support service discovery. For these reasons, we suggest combining the two protocols into a new protocol that integrates their best features into a feature-complete remote control protocol. The concept protocol relies on SOME/IP for discovery and initialization of the peripherals, while it uses IEEE1722 for streaming control messages. In the next sections, we provide the system model and operation, in addition to practical insights into the implementation of this protocol and the key design considerations, using the I2C bus as an example.

5.2 System model

Figure 5.1 presents the system model. We define two types of configuration:

- **RCP client to Endpoint communication**: Carried over SOME/IP. Handles service discovery, peripheral initialization, and configuration through the SOME/IP service.
- **RCP client to Device communication**: Carried over IEEE1722 control frames. The RCP node acts as a gateway tunneling operations sent from the RCP client to the device connected to an endpoint on the node, including operations such as I2C_transmit and I2C_transmit_receive.

Figure 5.3 illustrates the remote control node's operation. The operation is divided into 4

phases. Phase 1, Service Discovery, involves locating the devices connected to the node and their corresponding endpoints. Phase 2, Initialization, includes subscribing to an interface field or its corresponding service. Phase 3, Communication, the client uses discovery information from Phase 1 to establish direct communication with the IEEE1722 gateway. Finally, Phase 4, Teardown, involves terminating any subscriptions and terminating the communication. The service discovery and configuration mechanisms can be viewed as an additional layer on top of the gateway operation, which may be omitted in critical applications to improve performance. In this case, the peripherals are statically configured.

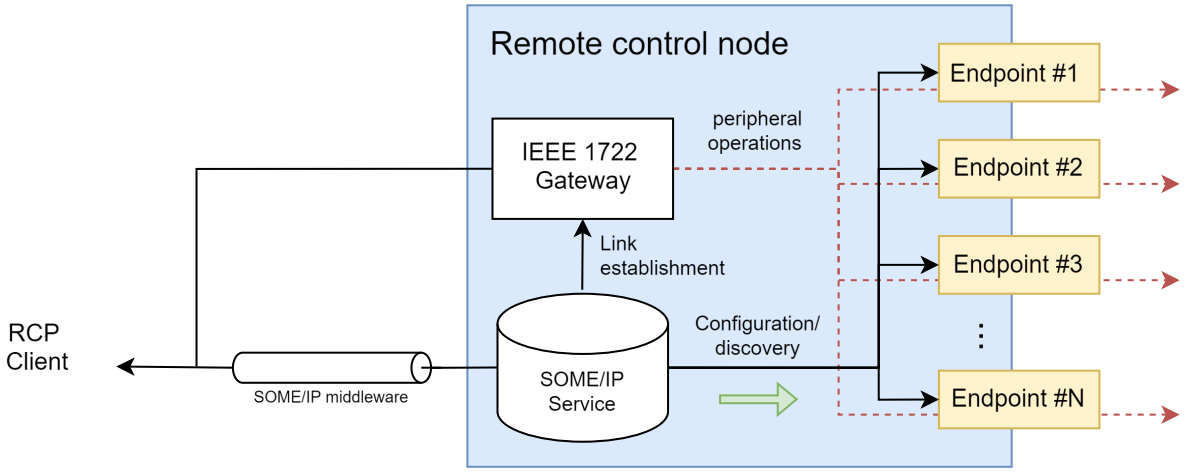


Fig. 5.1: RCP proposal system model

5.3 Protocol operation

In this section, we describe the operation of each phase, starting from the discovery phase and ending at the teardown phase following the communication flow presented in Figure 5.2.

5.3.1 Setup phase

Service discovery

At its core, service discovery allows the RCP client to locate endpoints and the node they belong to on the network. All devices are typically identified by a global ID (GID), which is unique across the network. The GIDs could be flashed onto the node during production of the RCP node, e.g., by Tier1 company or during integration by the OEM. With service discovery, the same device can be connected to any combination of nodes and remain discoverable on the network.

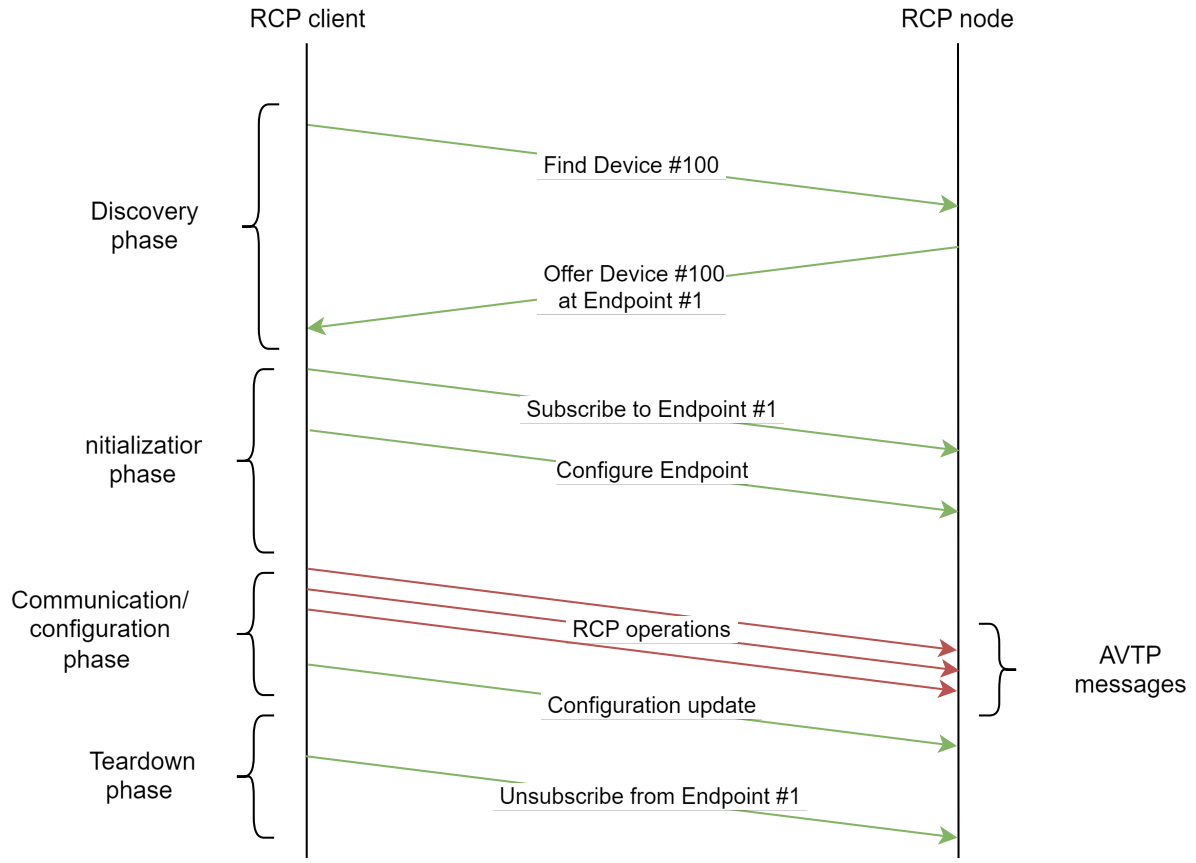


Fig. 5.2: RCP proposal communication flow

The implementation of the service discovery mechanism could vary, as the node could either implement a single service representing the node itself or implement multiple services such that each service is mapped to an endpoint. The latter solution is more straightforward to implement as endpoint IDs could be directly mapped to service IDs. Figure 5.3 shows an example service discovery scenario. The RCP nodes interface two devices with GIDs #100 and #101, which are respectively connected to endpoints #1 and #2. The discovery service(s) reads the local GIDs and broadcasts offer requests to the networking containing information on how to directly access the devices through the IEEE1722 gateway.

Peripheral initializaton

After the discovery, the RCP client can establish a connection with the SOME/IP service to configure the interface. Configuration parameters could be implemented as fields set/read by the client. Additionally, event notifications can be used for events such as interrupt signals and error handling.

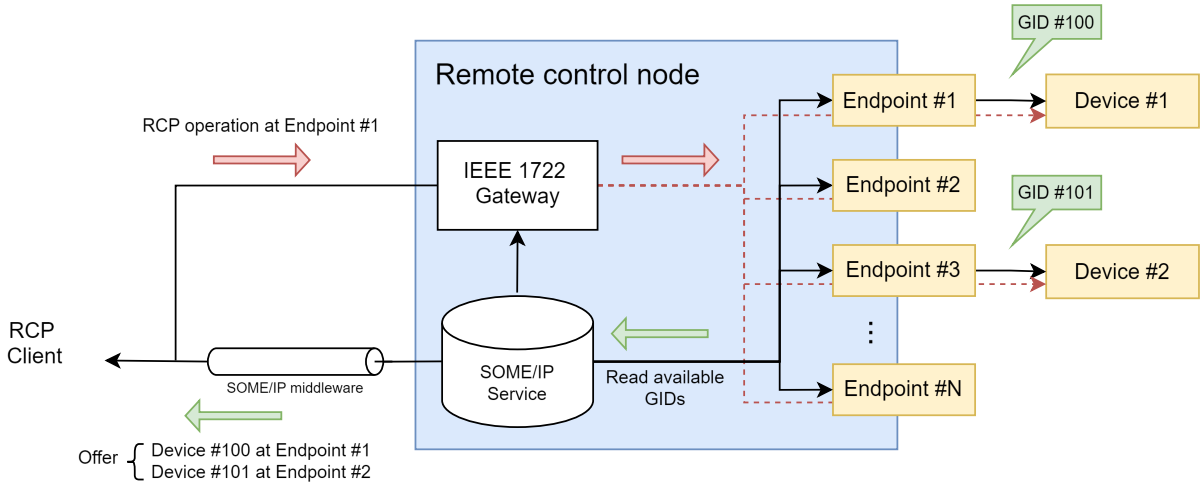


Fig. 5.3: RCP proposal operation and service discovery

5.3.2 Communication phase

AVTP operations

The client uses the identification parameters provided by the discovery mechanism to send RCP operations to the IEEE1722 gateway. For IEEE1722, these include the network address and other parameters, such as the bye bus ID in case of communication with a field. Additionally, the service could use the Stream Reservation Protocol (SRP) to reserve the necessary network resources for the IEEE1722 stream and provide the client with the stream ID.

Configuration updates

The client remains connected to the service throughout the whole communication process. It can still send configuration updates to the service if needed.

5.3.3 Teardown phase

Finally, to terminate communication, the client revokes its subscriptions to the service, and the service releases the reserved streams. Furthermore, the client can send a shutdown command to the service to power down the peripheral.

6 Conclusion and Future Prospects

Summary

This thesis presented the concept of remote control protocols and explored various remote control use cases, including ambient lighting control, camera streaming, and control, as well as a detailed low-level examination of I2C and SPI peripherals control. The use cases were used to extract a list of functional and non-function requirements for RCP. It presented two example use-cases for RCP: PWM/Motor driver control over I2C, demonstrating a low latency, low bandwidth control application, and OLED display control over SPI, representing a high bandwidth streaming application. Both use cases were implemented using state-of-the-art automotive communication protocols, SOME/IP and IEEE1722, candidate transport protocols for RCP. The thesis presented a detailed comparison between the two protocols based on the RCP requirements defined earlier.

The evaluations indicate that neither of the two protocols fully satisfies the requirements of RCP. IEEE1722 covers most of these requirements and is better suited for the streaming and control applications required by RCP. It supports control formats for the most common automotive peripherals and integrates well into time-sensitive networks. The evaluations of the two proposed use cases show that IEEE1722 outperformed SOME/IP in metrics such as latency, overhead, and load. However, SOME/IP was more favorable for implementing service-oriented communication and service discovery, which IEE1722 does not support. It became clear that while there is no clear winner, the two protocols complement each other's functionality.

Finally, the thesis provided a proposal for a new RCP that combines SOME/IP and IEEE1722 into a feature-complete remote control protocol. The concept protocol relies on SOME/IP to discover and initialize the peripherals, while it uses IEEE1722 to stream control messages. The thesis carefully detailed the interaction model of the system, its operation, and a description of its different phases.

Limitations and Future work

This thesis does not implement the use cases from initialization and configuration perspectives. Future work could focus on evaluating the startup times and scalability of the RCP system in more detail. Furthermore, evaluating protocols such as Data Distribution Service (DDS) and Zenoh, prominent protocols making their way to the automotive networks, and E2B, a proprie-

tary protocol, could provide additional insights on the design of remote control protocols.

bibliography

[1] Aptiv, "What Is a Software-Defined Vehicle?" March 2020. [Online]. Verfügbar: https://www.aptiv.com/en/insights/article/what-is-a-software-defined-vehicle [Zugriff am: 2024-12-16]

[2] R. Argolini, O. Burkacky, S. Johnston, S. Pellegrinelli, und G. Wachter, "Automotive software should costing: A new procurement tool for automotive companies | mckinsey," *McKinsey's Advanced Electronics Practice*, Sep. 2020. [Online]. Verfügbar: https://www.mckinsey.com/industries/industrials-and-electronics/our-insights/software-should-costing-a-new-procurement-tool-for-automotive-companies#/ [Zugriff am: 2024-12-16]

[3] G. Keßler, D. Sieben, A. Bhange, und E. Börner, "The Software Defined Vehicle – Technical and Organizational Challenges and Opportunities," in *23. Internationales Stuttgarter Symposium*, A. C. Kulzer, H.-C. Reuss, und A. Wagner. Wiesbaden: Springer Fachmedien, 2023, Seite 414–426.

[4] Z. Liu, W. Zhang, und F. Zhao, "Impact, Challenges and Prospect of Software-Defined Vehicles," *Automotive Innovation*, Band 5, Nr. 2, Seite 180–194, Apr. 2022. [Online]. Verfügbar: https://doi.org/10.1007/s42154-022-00179-z [Zugriff am: 2024-12-03]

[5] JamaSoftware, "Software defined vehicles: Revolutionizing the future of transportation." [Online]. Verfügbar: https://www.jamasoftware.com/whitepaper/software-defined-vehicles-revolutionizing-the-future-of-transportation-whitepaper [Zugriff am: 2025-01-07]

[6] B. Canis und R. K. Lattanzio, "U.S. and EU Motor Vehicle Standards: Issues for Transatlantic Trade Negotiations," Feb. 2014. [Online]. Verfügbar: https://crsreports.congress.gov/product/pdf/R/R43399 [Zugriff am: 2025-03-01]

[7] V. Bandur, G. Selim, V. Pantelic, und M. Lawford, "Making the Case for Centralized Automotive E/E Architectures," *IEEE Transactions on Vehicular Technology*, Band 70, Nr. 2, Seite 1230–1245, Feb. 2021. [Online]. Verfügbar: https://ieeexplore.ieee.org/document/9337216/ [Zugriff am: 2024-07-05]

[8] futuremobilitymedia, "SUBSCRIPTIONS AND THE SDV: Where is the value proposition," Sep. 2024. [Online]. Verfügbar: https://futuremobilitymedia.com/subscriptions-and-the-sdv-where-is-the-value-proposition/ [Zugriff am: 2025-01-07]

[9] S. Jiang, "Vehicle E/E Architecture and Its Adaptation to New Technical Trends." SAE International, Apr. 2019. [Online]. Verfügbar: https://saemobilus.sae.org/papers/vehicle-e-e-architecture-adaptation-new-technical-trends-2019-01-0862 [Zugriff am: 2025-01-08]

[10] W. Wang, K. Guo, W. Cao, H. Zhu, J. Nan, und L. Yu, "Review of Electrical and Electronic Architectures for Autonomous Vehicles: Topologies, Networking and Simulators," *Automotive Innovation*, Band 7, Nr. 1, Seite 82–101, Feb. 2024. [Online]. Verfügbar: https://doi.org/10.1007/s42154-023-00266-9 [Zugriff am: 2024-12-16]

[11] Aptiv, "Evolution of Vehicle Architecture," June 2018. [Online]. Verfügbar: https://www.aptiv.com/en/insights/article/evolution-of-vehicle-architecture [Zugriff am: 2024-12-16]

[12] H. Zhu, W. Zhou, Z. Li, L. Li, und T. Huang, "Requirements-Driven Automotive Electrical/Electronic Architecture: A Survey and Prospective Trends," *IEEE Access*, Band 9, Seite 100 096–100 112, 2021, conference Name: IEEE Access. [Online]. Verfügbar: https://ieeexplore.ieee.org/abstract/document/9466854 [Zugriff am: 2024-08-30]

[13] P. Hank, T. Suermann, und S. Müller, "Automotive Ethernet, a Holistic Approach for a Next Generation In-Vehicle Networking Standard," in *Advanced Microsystems for Automotive Applications 2012*, G. Meyer. Berlin, Heidelberg: Springer, 2012, Seite 79–89.

[14] *10BASE-T1S System Implementation Specification*, OPEN ALLIANCE, 2023, 1.0. [Online]. Verfügbar: https://opensig.org/wp-content/uploads/2023/12/20230215_10BASE-T1S_system_implementation_V1_0.pdf

[15] "Why use 10BASE-T1S instead of CAN?" [Online]. Verfügbar: https://www.keysight.com/blogs/en/tech/2024/02/8/how-is-10base-t1s-different-from-can [Zugriff am: 2025-02-14]

[16] I. . L. , "TSN for Automotive." [Online]. Verfügbar: https://www.ieee802.org/1/files/public/docs2024/admin-tsn-automotive-flyer-1124.pdf [Zugriff am: 2025-02-14]

[17] "Ieee standard for local and metropolitan area networks–timing and synchronization for time-sensitive applications," *IEEE Std 802.1AS-2020 (Revision of IEEE Std 802.1AS-2011)*, Seite 1–421, 2020.

[18] "Ieee standard for local and metropolitan area networks–virtual bridged local area networks amendment 14: Stream reservation protocol (srp)," *IEEE Std 802.1Qat-2010 (Revision of IEEE Std 802.1Q-2005)*, Seite 1–119, 2010.

[19] "Ieee standard for a transport protocol for time-sensitive applications in bridged local area networks," *IEEE Std 1722-2016 (Revision of IEEE Std 1722-2011)*, Seite 1–233, 2016.

[20] "Ieee standard for a transport protocol for time-sensitive applications in bridged local area networks." [Online]. Verfügbar: https://standards.ieee.org/ieee/1722/5979/ [Zugriff am: 2025-02-11]

[21] "Ieee p1722b draft 1.8 contribution – i2c." [Online]. Verfügbar: https://grouper.ieee.org/groups/1722/contributions/2022/1722b-I2C-contrib-bwy-v10-2022-10-11_RevAfterFinal.pdf [Zugriff am: 2025-02-11]

[22] "Requirements on IEEE1722." [Online]. Verfügbar: https://www.autosar.org/fileadmin/standards/R24-11/FO/AUTOSAR_FO_RS_IEEE1722.pdf [Zugriff am: 2025-02-11]

[23] L. Völker, "Scalable service-Oriented MiddlewarE over IP (SOME/IP)." [Online]. Verfügbar: http://some-ip.com/ [Zugriff am: 2025-02-11]

[24] *SOME/IP Protocol Specification*, AUTOSAR, r22-11. [Online]. Verfügbar: https://www.autosar.org/fileadmin/standards/R22-11/FO/AUTOSAR_PRS_SOMEIPProtocol.pdf

[25] *SOME/IP Service Discovery Protocol Specification*, AUTOSAR, r22-11. [Online]. Verfügbar: https://www.autosar.org/fileadmin/standards/R22-11/FO/AUTOSAR_PRS_SOMEIPServiceDiscoveryProtocol.pdf

[26] *Specification on SOME/IP Transport Protocol*, AUTOSAR, r21-11. [Online]. Verfügbar: https://www.autosar.org/fileadmin/standards/R21-11/CP/AUTOSAR_SWS_SOMEIPTransportProtocol.pdf

[27] "What is an endpoint? | Endpoint definition." [Online]. Verfügbar: https://www.cloudflare.com/learning/security/glossary/what-is-endpoint/ [Zugriff am: 2025-02-03]

[28] H. Senninger, "Car makers to use intelligent ambient lighting to create new functions – and a new feeling – inside the cabin," Oct. 2023. [Online]. Verfügbar: https://ams-osram.com/news/blog/car-makers-to-use-intelligent-ambient-lighting-to-create-new-functions-and-a-new-feeling-inside-the-cabin [Zugriff am: 2025-01-26]

[29] M. Hassib, M. Braun, B. Pfleging, und F. Alt, "Detecting and Influencing Driver Emotions Using Psycho-Physiological Sensors and Ambient Light," in *Human-Computer Interaction – INTERACT 2019*, D. Lamas, F. Loizides, L. Nacke, H. Petrie, M. Winckler, und P. Zaphiris. Cham: Springer International Publishing, 2019, Seite 721–742.

[30] F. Hurley und P. Wilms, "Centralized software and zonal architectures for future innovative ambient lighting enabled by e2b 10base-t1s," Nov. 2022. [Online]. Verfügbar: https://www.analog.com/media/en/news-marketing-collateral/solutions-bulletins-brochures/centralized-software-and-zonal-architectures-e2b-10base-t1s.pdf [Zugriff am: 2024-08-30]

[31] "ISELED Protocol Ambient Lighting." [Online]. Verfügbar: https://www.microchip.com/en-us/solutions/automotive-and-transportation/body-electronics/iseled-protocol-ambient-lighting [Zugriff am: 2025-01-26]

[32] *Optimized all Hardware Edge Node, 10BASE-T1S Ethernet to the Edge Bus (E2B) Transceiver*, Analog Devices, 2024, rev. Sp0. [Online]. Verfügbar: https://www.analog.com/media/en/technical-documentation/data-sheets/ad3300-01-04-05.pdf

[33] L. L. Bello, "Novel trends in automotive networks: A perspective on Ethernet and the IEEE Audio Video Bridging," in *Proceedings of the 2014 IEEE Emerging Technology and Factory Automation (ETFA)*, Sep. 2014, Seite 1–8, iSSN: 1946-0759. [Online]. Verfügbar: https://ieeexplore.ieee.org/document/7005251/?arnumber=7005251 [Zugriff am: 2025-01-26]

[34] C. Ruth, "Ethernet Camera Bridge for Software-Defined Vehicles," Jan. 2023, section: Beyond Standards. [Online]. Verfügbar: https://standards.ieee.org/beyond-standards/ethernet-camera-bridge-for-software-defined-vehicles/ [Zugriff am: 2024-07-09]

[35] *Automotive 2-MP Camera Module Reference Design With MIPI CSI-2 Video Interface, FPD-Link III and POC*, Texas Instruments, 2018. [Online]. Ver-

fügbar: https://www.ti.com/lit/ug/tidud51a/tidud51a.pdf?ts=1737931035346&ref_url=https%253A%252F%252Fduckduckgo.com%252F

[36] H. Hsiang, K.-C. Chen, P.-Y. Li, und Y.-Y. Chen, "Analysis of the Effect of Automotive Ethernet Camera Image Quality on Object Detection Models," in *2020 International Conference on Artificial Intelligence in Information and Communication (ICAIIC)*, Feb. 2020, Seite 021–026. [Online]. Verfügbar: https://ieeexplore.ieee.org/document/9065232/ [Zugriff am: 2025-01-27]

[37] *I2C-bus specification and user manual*, NXP Semiconductors, 2021, rev. 7.0. [Online]. Verfügbar: https://www.nxp.com/docs/en/user-guide/UM10204.pdf

[38] *STM32H745/755 and STM32H747/757 advanced Arm ® -based 32-bit MCUs*, STMicroelectronics, 2023, rev 4. [Online]. Verfügbar: https://www.st.com/resource/en/reference_manual/rm0399-stm32h745755-and-stm32h747757-advanced-armbased-32bit-mcus-stmicroelectronics.pdf

[39] *Description of STM32H7 HAL and low-layer drivers*, STMicroelectronics, 2022, rev 6. [Online]. Verfügbar: https://www.st.com/resource/en/user_manual/um2217-description-of-stm32h7-hal-and-lowlayer-drivers-stmicroelectronics.pdf

[40] *Using the Serial Peripheral Interface to Communicate Between Multiple Microcomputers*, NXP Semiconductors, 2002, rev. 1. [Online]. Verfügbar: https://www.nxp.com/docs/en/application-note/AN991.pdf

[41] M. Rumez, D. Grimm, R. Kriesten, und E. Sax, "An Overview of Automotive Service-Oriented Architectures and Implications for Security Countermeasures," *IEEE Access*, Band 8, Seite 221 852–221 870, 2020. [Online]. Verfügbar: https://ieeexplore.ieee.org/document/9285284/ [Zugriff am: 2024-07-05]

[42] "Home - Open Alliance," Dec. 2023. [Online]. Verfügbar: https://opensig.org/ [Zugriff am: 2025-02-14]

[43] R. P. Ltd, "Raspberry Pi 4 Model B." [Online]. Verfügbar: https://www.raspberrypi.com/products/raspberry-pi-4-model-b/ [Zugriff am: 2025-02-11]

[44] ——, "Raspberry Pi 5." [Online]. Verfügbar: https://www.raspberrypi.com/products/raspberry-pi-5/ [Zugriff am: 2025-02-11]

[45] "Welcome to The Linux PTP Project." [Online]. Verfügbar: https://linuxptp.nwtime.org/ [Zugriff am: 2025-02-11]

[46] "Overview of IEEE 802.1AS Generalized Precision Time Protocol (gPTP) — ECI documentation." [Online]. Verfügbar: https://eci.intel.com/docs/3.0/development/tsnrefsw/tsn-overview.html [Zugriff am: 2025-02-11]

[47] *PTP4l (8)*, Debian, 2023, linux User Manual. [Online]. Verfügbar: https://manpages.debian.org/unstable/linuxptp/ptp4l.8.en.html

[48] *clock_gettime (3)*, Linux man-pages project, 2020, linux Programmer's Manual. [Online]. Verfügbar: https://www.man7.org/linux/man-pages/man3/clock_gettime.3.html

[49] *time(7)*, Linux man-pages project, 2024, linux Programmer's Manual. [Online]. Verfügbar: https://www.man7.org/linux/man-pages/man7/time.7.html

[50] "Flexible resolution oscilloscope specifications." [Online]. Verfügbar: https://www.picotech.com/oscilloscope/5000/picoscope-5000-specifications/ [Zugriff am: 2025-02-11]

[51] "ProfiShark 1G+ | Gigabit Ethernet TAP with GPS Features | Profitap." [Online]. Verfügbar: https://www.profitap.com/profishark-1g-plus/ [Zugriff am: 2025-02-11]

[52] *PCA9685 16-channel, 12-bit PWM Fm+ I2C-bus LED controller*, Adafruit, 2015, rev. 4. [Online]. Verfügbar: https://cdn-shop.adafruit.com/datasheets/PCA9685.pdf

[53] M. Iorio, A. Buttiglieri, M. Reineri, F. Risso, R. Sisto, und F. Valenza, "Protecting In-Vehicle Services: Security-Enabled SOME/IP Middleware," *IEEE Vehicular Technology Magazine*, Band 15, Nr. 3, Seite 77–85, Sep. 2020. [Online]. Verfügbar: https://ieeexplore.ieee.org/document/9085373/ [Zugriff am: 2024-07-09]

[54] "1.5inch OLED Module - Waveshare Wiki." [Online]. Verfügbar: https://www.waveshare.com/wiki/1.5inch_OLED_Module#Hardware_Interface [Zugriff am: 2025-02-11]

[55] "EV08L38A." [Online]. Verfügbar: https://www.microchip.com/en-us/development-tool/ev08l38a [Zugriff am: 2025-02-11]

Abbreviations

Abbreviation	Meaning
ACF	AVTP Control Format
ADAS	advanced driver assistance system
AVB	Audio/Video Bridging
AVTP	Audio Video Transport Protocol
AAF	AVTP Audio Format
BMCA	Best Master Clock Algorithm
CAN	controller area network
CCU	connectivity control unit
CPU	central processing unit
EEA	electrical and electronic architecture
ECU	electronic control unit
E2B	Ethernet to the Edge Bus
FQTSS	Forwarding and Queuing Enhancements for Time-Sensitive Streams
GBB	Generic Byte Bus
GM	Grand Master
gPTP	generalized Precision Time Protocol
IEEE	Institute of Electrical and Electronics Engineers
IVN	in-vehicle network
LIN	local interconnect network
MOST	media oriented system transport
NHTSA	National Highway Traffic Safety Administration
NTSCF	Non-Time-Synchronous Control Format
NIC	network interface card
OEM	original equipment manufacturer
OS	operating system
OTA	over the air
RCP	remote control protocol
RTT	round-trip time

Abbreviation	Meaning
SDV	software defined vehicle
SOA	service oriented architecture
SOME/IP	Scalable service-Oriented MiddlewarE over IP
SOME/IP-SD	Scalable service-Oriented MiddlewarE over IP - Service Discovery
SOME/IP-TP	Scalable service-Oriented MiddlewarE over IP - Transport Protocol
SPI	serial peripheral interface
SRP	Stream Reservation Protocol
TSCF	Time-Synchronous Control Format
V2C communication	vehicle-to-cloud communication
V2I communication	vehicle-to-infrastructure communication
V2V communication	vehicle-to-vehicle communication
TSN	time-sensitive networks

Tabellenverzeichnis

Abbildungsverzeichnis