FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

Rust-based SOME/IP implementation for robust automotive software

João Francisco Barreiros de Almeida

Mestrado Integrado em Engenharia Informática e Computação

Supervisor: Luís Miguel Pinho de Almeida

Co-supervisor: Michael Breitung

2nd co-supervisor: Stefan Boiciuc

March 7, 2021

Rust-based SOME/IP implementation for robust automotive software

João Francisco Barreiros de Almeida

Mestrado Integrado em Engenharia Informática e Computação

March 7, 2021

Resumo

Os veículos modernos contêm um número de subsistemas electrónicos e informáticos que está constantemente a crescer. Esta situação resulta num aumento significativo da complexidade do sistema para a integração de todos esses sub-sistemas com requisitos crescentes quer computacionais quer de largura de banda e de proteção contra interferências mûtuas e externas. Deste modo, a indústria automóvel está a adoptar comunicações com base em Ethernet e os protocolos de comunicação TCP/IP. Esta transição promoveu a adoção de uma Arquitetura Orientada a Serviços, permitindo uma maior flexibilidade, escalabilidade e reusabilidade dos componentes de *hardware* e de *software* do veículo. Seguindo esta tendência, a BMW criou o *Scalable service-Oriented Middleware over IP* (SOME/IP), um *middleware* apto para vários tipos de dispositivos e sistemas operativos, usando como base os protocolos TCP/IP, havendo portanto uma sinergia com Ethernet. Por sua vez, a aliança GENIVI que fornece soluções *open-source* para os *use cases* automóveis, incorporou o SOME/IP na sua CommonAPI, uma API em C++ de abstração do *middleware*. O C++ em conjunto com C dominam a indústria de sistemas embarcados, contudo, para desenvolver código seguro é necessário seguir um conjunto de regras de desenvolvimento. Quando a Mozilla introduziu o Rust no mundo da programação gerou interesse dos programadores por ter características similares a C/C++ mas providenciando um robusto analizador estático de código no compilador que deteta várias classes de erros permitindo assim uma segurança de código mais elevada. O objectivo é implementar as ferramentas GENIVI que são usadas no desenvolvimento de um sistema com base no SOME/IP, oferecendo ao programador a opção de utilizar um ambiente Rust mantendo a interoperabilidade com as implementações existentes. Este projeto começará com a análise do protocolo básico, estrutura das mensagens e *Service Discovery*, subindo para as abstrações da CommonAPI e finalmente a geração de código Rust com base na *Franca Interface Definition Language* (Franca IDL). No fim, com a comparação entre a nossa implementação e a da GENIVI mostramos como o principal objectivo deste trabalho, a interoperabilidade, foi alcançado. O resultado é uma implementação que cumpre o que foi previsto neste trabalho mas que ainda necessita de refinamentos adicionais para ser tão completa como a implementação de referência.

Abstract

Modern vehicles possess an ever growing number of electronics and software components to support functions that range from vehicle confort system, such as interior climate, ventillation and seat adjustment, to Advanced Driving Assistance Systems or for information and entertainment functions. This situation results in a significant increase of the system complexity to integrate all those sub-systems with increasing computing and communication requirements, as well as protection against mutual and external interference. Thus the automotive industry is moving towards the virtualization of the vehicle hardware and software platform, relying on Ethernet connectivity and TCP/IP communication protocols. This transition promoted the adoption of a Service-oriented Architecture, allowing higher flexibility, scalability and reusability of the vehicle components. Existing Service Oriented Architecture solutions could not be used in the automotive domain as they did not meet the real-time requirements of a vehicle. Consequently, BMW designed Scalable service-Oriented Middleware over IP (SOME/IP), a middleware fit for all kinds of devices and operating systems, built on top of the TCP/IP protocols, matching well with Ethernet. In turn, the GENIVI Alliance, who provides open-sourced solutions for the automotive use cases, incorporated it in its CommonAPI, a C++ abstraction API independent of the middleware. C++ alongside C dominate the embedded system industry, however, to develop safe code it is necessary to follow a set of development guidelines. When Mozilla introduced Rust in the programming world it sparked developers interest for having similar characteristics to C/C++ but offering a robust built-in code analysis in the compiler that detects several classes of errors thus providing a much higher code safety level. The objective is to implement the GENIVI tools for the development of a SOME/IP system using the Rust language, allowing a developer to fully utilize a Rust environment while maintaining interoperability with existing implementations. The work will start from the SOME/IP protocol and its Service Discovery module, building up to the CommonAPI SOME/IP bindings and finally generate Rust code based on the Franca Interface Definition Language (Franca IDL). In the end we will compare our implementation and the one from GENIVI, showing that the main aspect, interoperability, was achieved. The result is an implementation that matches what was proposed in this work but still requires additional effort to make it as complete as the reference implementation.

Glossary

API Application Interface

AUTOSAR Automotive Open System Architecture

AVB Audio Video Bridging

CAN Controller Area Network

ECU Electronic Control Unit

Franca IDL Franca Interface Definition Language

GPU Graphics Processing Unit

IEEE Institute of Electrical and Electronics Engineers

IVI In-Vehicle Infotainment

IVN In-Vehicle Network

JSON JavaScript Object Notation

OS Operating System

RPC Remote Procedure Call

SOA Service-oriented Architecture

SOME/IP Scalable service-Oriented Middleware over IP

SOME/IP-SD Service Oriented Middleware over IP Service Discovery

TCP Tranmission Control Protocol

TSN Time-Sensitive Networking

UDP User Datagram Protocol

Acknowledgements

First and foremost, I would like to express my gratitude to the supervisors that directly assisted me in this work, Luís Almeida, Michael Breitung and Stefan Boiciuc. Even during an unprecedented year where everyone was relearning how to efficiently do their jobs they supported me throughout the year. Without their help this document would be in a much worse state.

A special thanks also goes out to Elektrobit Automotive GmbH for taking me in and accompanying me in my final months as a student. Praise also needs to be given to all of the nameless colleagues that handled the bureaucratic headache that was required in order for this work to be executed. Lastly, a more affectionate thanks to my family and friends that supported me in many ways throughout this journey.

João Francisco Barreiros de Almeida

Chapter 1

Introduction

Ever since their conception phase, in the 19th century, automobiles have had a unique set of requirements. As a technology that can put citizens in danger, it needs to be safe, reliable and robust while operating under a wide range of conditions. The first cars to be built used purely mechanical components with no need for software. After the first programmable electronic component was used inside a car it inaugurated the introduction of software as an intrinsic element of an automobile. As more functionalities were added to the vehicle, with their corresponding software, a single ECU did not have enough memory or computing power to control all of the system and subsystems of a single vehicle, it became necessary to have multiple ECUs communicate with one another, thus an internal communications network had to be used.

Before any protocol was standardized between the different suppliers, each had their own specific network protocol and topology. These protocols, alongside the functionality of ECUs, was all initially developed using only Assembly. As the set of software development tools kept improving, several programming languages would surface with varying degrees of adoption. One of the programming languages that appeared was C. It stood out for its versatility and portability while still providing the developer with low-level access to the system memory. As a language that effectively abstracted the development of software from the target system, it quickly became the language of choice in the embedded software development industry and, to this date, is still the most used language for embedded software [38].

Alongside this adoption of C, a standardization of the automobile internal network was also taking place. Seeing that these types of networks had a unique set of requirements, the designation vehicle bus was given. One of the first developed, and the one that became ubiquitous in the automotive industry was CAN, a message based protocol designed for multiplex electrical wiring within automobiles.

Incorporating the portability of C and the standardization of CAN, the pillars were built for the expansion of the automotive software industry. This meant an increasing amount of ECUs was being inserted in a single automobile. Since different ECUs have different architectures, even when produced by the same supplier, too much effort was spent dealing with each component intricacies instead of the actual design of the system. Hence, in 2003, the automotive industry decided

to create a standardization initiative named Automotive Open System Architecture (AUTOSAR) with the goal of abstracting the hardware layer and facilitating programming and interoperability between different ECUs.

In more recent years, additional vehicle buses were introduced, such as FlexRay, MOST and LIN, each with their own set of characteristics. All of these protocols provided even greater flexibility for the developers, so more functionalities kept getting introduced in an automobile, translating into an unmanageable code base. To cope with this growth, most of the non-safety-critical components have moved to more recent technologies, commonly C++, which allows seamless integration with existing C code while maintaining software efficiency. By providing an Object-Oriented Architecture, code with related purposes can be encapsulated in a class which provides higher modularity and lower interdependencies, effectively encouraging cleaner and reusable software. As such, complex components are now present in a car, such as LIDAR sensors and High-Resolution cameras, that by far surpass the maximum data throughput of traditional vehicle buses.

To meet these new transmission rates, the industry showed interest in the already established and widespread Ethernet technology that allows rates of up to 100Gb/s. Using Ethernet inside an automobile also presents the opportunity for the automotive software to match with the ubiquitous TCP/IP protocol and follow a Service-oriented Architecture (SOA) design. In 2011, BMW introduced the first SOA automotive protocol, Scalable service-Oriented Middleware over IP (SOME/IP) supporting many of the concepts that were already familiar to SOA developers. A few years later, in 2016, SOA was also introduced in AUTOSAR as the Adaptive Platform using SOME/IP as its first supported communication protocol. Nowadays, SOME/IP is mostly used as the protocol for the infotainment systems and other non-safety-critical functionalities.

1.1 Motivation

Most of the embedded software world is developed using a combination of C and C++. The amount of control that is given to the developer in these languages is a double-edged sword. This control allows the developers to perfectly fine-tune the system to meet its requirements. However, there are numerous errors and unintended behavior that may happen if the developer is not careful enough. Some of these errors may even only be discovered once the software is in production, which can pose a severe safety threat.

To avoid this, programming guidelines have been designed for both C and C++ that when followed facilitate the development of safe, secure and reliable code. The one adopted even by AUTOSAR is MISRA, that contains both a C and a C++ version. Its rules range from simple coding styles to the total prohibition of undefined behavior. While these rules certainly help and are of utmost interest, there is no simple way to follow them. Undefined behavior is omnipresent in both C/C++ as a misuse of even a simple statement can cause it. The specification of C, which is considered a simple language without many of the features of C++, contains over 800 pages as of C11. It is not reasonable to expect a developer to be fully aware of that specification, so errors in development are to be expected.

Mozilla has tackled this issue with the development of the Rust programming language. Ever since its appearance in 2010, it has been gaining developers interest. It has peaked in more recent years, being the most loved programming language since 2016 according to Stack Overflow Developer Surveys. Rust allows a control that is similar to C and C++ but enforces various memory safety rules at compile time. This drastically reduces the number of errors that are caused due to mishandling of memory, which is often the reason for undefined behavior.

1.2 Objectives

The aim of this work is to provide a Rust environment when developing software using SOME/IP as the underlying middleware. Our implementation will be based on the tools provided by the GENIVI alliance, which is also hosting the original SOME/IP done by BMW named *vsomeip*. Apart from the actual protocol, GENIVI uses the CommonAPI to provide further abstraction from the underlying protocol. Currently, CommonAPI provides support for D-Bus and SOME/IP. In this project, D-Bus will be ignored, and the sole focus will be the SOME/IP part. A usual workflow would be to first design the service interfaces of the system. CommonAPI uses the Franca Interface Definition Language (Franca IDL) for this specification. Afterwards, code is generated based on the service interfaces, which is already linked with the CommonAPI library and the underlying middleware. A developer would then start building its application based on the developed code. As such, our implementation will start with the base SOME/IP protocol, including the Service Discovery module (SOME/IP-SD), followed by the CommonAPI library and finally the code generator. These three tools should allow a developer to fully develop an application using Rust while still maintaining interoperability with the original implementations.

1.3 Structure

The structure of the chapters of this document is as follows:

2. **Background**

   Provides insight into the different technologies and standards that have shaped the automotive software industry. The analysis starts with the more hardware related Institute of Electrical and Electronics Engineers (IEEE) standards and then into the automotive software platforms and standards that build on top of the IEEE standards. At the end of the chapter, an overview of the state of the art automotive software technologies and the software industry initial adoption of Rust for a broad set of scopes.

3. **Reference toolchain analysis**

   Gives an overview of the whole CommonAPI library and how it is used to develop applications that are abstracted from the underlying middleware. It starts by analyzing the code generators, then moves to the CommonAPI Runtime and finally GENIVI SOME/IP implementation, `vsomeip`.

4. **Rust implementation**

   Contains a description of each of the components implemented in this work and how they differ from the reference implementation. We follow a top-down analysis, beginning at the code generator, then our CommonAPI Runtime implementation and finally the SOME/IP implementation. For each of those components, an initial comparison between the reference implementation and the Rust implementation is also given, alongside with the shortcomings of the Rust implementation at the end of each section.

5. **Implementation validation**

   Highlights the final validation of the project done in this work. Initially, a demonstration of a developed system is shown which can also integrate with a C++ application. Then an overview of the integration tests that were used to validate the behavior of the Rust SOME/IP implementation. Lastly, a comparison of the performance between the Rust and the reference implementation is made in the last section using different scenarios, alongside an explanation of the differing results.

6. **Conclusion**

   As the last chapter, it gives some final remarks about the work done, an overview of the results achieved, and what future work could be done on top of what was achieved here.

Chapter 2

Background

State of the art automobiles are now using the Ethernet protocol for connectivity between components that require it. Nonetheless, this adoption did not happen overnight, an adaptation of the existing protocol had to be done for it to be applied to the automotive domain. This chapter will provide insight into the adoption of Ethernet as a backbone automotive ECU communication protocol, from the reason as to why it was not used earlier to the automotive technologies that build on top of the Ethernet protocol to provide a robust and reliable automotive network.

2.1 Automotive Ethernet

Ethernet is a technology that is widely used throughout the world. Standardized in 1983, it has continuously been evolving to support higher bit rates and longer link distances. While it has existed for a long time, only recently has the automotive industry shown interest in Ethernet to replace traditional vehicle buses. The reason for this is that it could not meet the hard-timing requirements of an automotive network, namely latencies in the low microsecond range could not be guaranteed, which is needed for very fast reactors, for example the Anti-lock Braking System. Protection from Electromagnetic Interference and Radio Frequency Interference was also insufficient, resulting in too much noise in the data being transmitted. Furthermore, Ethernet did not provide a way to synchronize time between different devices, leaving that responsibility to the application.

One of the initial markets that drove the need for Ethernet networks with improved reliability, synchronization and very low latencies was the analog audio and video equipment. These devices have softer time constraints when compared to the automotive environment, but failure to meet them results in degraded user experience. To resolve these issues, the Audio Video Bridging (AVB) task group was formed by the IEEE. It aimed to develop a set of technologies and standards to allow time-sensitive communications over Ethernet. Shortly after its appearance it was renamed as Time-Sensitive Networking (TSN), as a broader range of industries showed interest in such standards, including the automotive industry.

2.1.1 Time-Sensitive Networking

Maintaining the original goal of the AVB task group but now with a much wider scope, TSN mainly defines extensions and enhancements to the IEEE 802.1Q - Bridges and Bridged Networks standard, that describes Local Area Networks (LAN) on an Ethernet network but allowing deterministic services. Moreover, it also contains profiles for different industries, encompassing a set of standards that meet the specific use cases of that industry. One of the profiles that is being developed is for automotive in-vehicle communications.

2.1.1.1 TSN for Automotive In-Vehicle Ethernet Communications

Given the range of conditions under which an automobile is expected to be fully functional, it made sense to create a profile of TSN specific for the automotive In-Vehicle Networks (IVN). This profile represents the state of the art specifications for IVN, and currently, there is not a final proposal, merely drafts. Considering that the suite of TSN standards is broad and intended for use in multiple environments that require bounded latency, high reliability and security, this profile specifies the features that are directly applicable to the automotive sector. Automotive IVN are unlike many other networks in the sense that every device is known when the network is being designed. Not only this but also the topology of the network is known and can only be modified in ways that were planned by the designers. Both of these facts allow network designers to narrow the selection of TSN standards [13].

As an industry that has a wide set of technologies that can be used, interoperability between them is required, so it comes naturally that Ethernet needs to interface with non-Ethernet networking technologies. Likewise, Ethernet technologies must be at least as robust as previous networking technologies throughout the whole life cycle of the vehicle. Thus it is necessary to support various kinds of failures in the network and still be able to provide the same quality of service. Should these failures persist, as in the case of a damaged wire, then the vehicle needs to be repaired. The existence of a maintenance mode greatly simplifies this process as it provides insight into the components of the network, allowing a quicker diagnosis of the exact problem in the vehicle.

2.1.1.2 IEEE 802.1CB: Frame Replication and Elimination for Reliability (FRER)

This standard describes techniques to configure redundancy through at least two network paths to ensure network reliability. By replicating packets, sending them through different paths on the network and then using the sequence number to eliminate duplicates, the probability of packet loss is reduced. There are mainly 5 functions provided, separated in layers, although only a subset might be needed depending on the application.

The first function, located at the top of the FRER layer is the sequencing function. Its responsibility is both to assign a sequence number to packets passed down the stack towards the physical layer and to use the sequence number from packets passed up the stack from multiple sources to decide whether to keep or discard them. The stream splitting function replicates the packets passed

down the stack assigning each a different path on the network, packets passed up the stack remain unchanged. The individual recovery function is similar to the sequencing function, only that it is only applied to a single source. This is to avoid congestion when a failure in one of the paths leads to a full or partial merging of different paths. The sequence decode/encode function inserts the sequence number into the actual packet passed down the stack and extracts the sequence number from packets passed up the stack. Finally, the stream identification function that is only applied to packets passed up the stack in order to identify through which path was the packet received [24].

These layers combined form a key component of the TSN automotive profile. IEEE 802.1CB gives the confidence that once a packet starts being transmitted that the receiving end will indeed receive it without the need for retransmission. The amount of redundancy that FRER uses can be configured. Therefore functionalities that require a packet to arrive with little margin for failure can use multiple redundant paths at the expense of having a more congested network [13]. Furthermore, through this standard, the manufacturers already have an intrinsic fail-safe mechanism since even if there is a problem with a certain wiring, the network can automatically detect this and readjust its redundant paths to not use the broken wire.

2.1.1.3 IEEE 802.1AS: Timing and Synchronization for Time-Sensitive Applications

IEEE 802.1AS provides protocols to ensure that the time is synchronized between the different nodes of the network. There will be a single node assigned as Grandmaster, which will serve as the whole network timing reference. However, as a fail-safe mechanism, the network is aware of other potential Grandmasters to minimize downtime on failure, although on normal operation these act as any non-Grandmaster nodes. Non-Grandmaster nodes can be further divided into two types, either a time-aware end node or a time-aware bridge node. End nodes serve merely as a recipient of the time information, bridge nodes receive time information directly or indirectly, through other bridge nodes, from the grandmaster and apply corrections to compensate for the transmission delays [19].

Applying this protocol to the automotive domain is made simpler when considering the assumption that the network topology and devices is known and will only change in ways already predicted. Thus it is possible to disable the Grandmaster selection process and manually assign it. Additionally, it might be necessary to have predefined fail scenarios where the Grandmaster switch is much faster than the process used in 802.1AS. Calculation of the delays between nodes is an aspect that can be calculated at design time and could be pre-configured in all devices. However, due to the possibility of a repair shop using a different component than the one specified by the manufacturer, the calculation of delays still needs to happen [13]. Finally, car manufacturers are also free to change the rate at which synchronization messages are sent through the network. The default value is 8 per second but can be modified to faster rates for quicker synchronization or slower rates that reduce network traffic and processing overhead.

2.1.1.4 IEEE 802.1Qci-2017: Per-Stream Filtering and Policing

IEEE 8021.Qci standard filters and policies individual traffic based on a set of rules. It prevents the overload and congestion of the network that can be caused either by erroneous packets due to a malfunction of some component or by a deliberate attack to the network. The rules for this filtering and policing are applied to individual streams and may overwrite the filtering and policing parameters of a packet, so even if a packet has the highest priority, it can be given the lowest priority once the policing is done. Furthermore, a restraint on the amount of bandwidth a single source is allowed to generate can be placed, further impairing the source of congesting the network [20].

By applying this standard to the automotive industry errors such as the blabbing idiot, where a component is generating more high priority traffic than it should be due to an error, are completely mitigated, and the network can continue operating normally. External attacks, such as Denial of Service attacks, are also mitigated. Since the topology of the network is known, the entry point for these attacks is also known. Thus the manufacturer can configure IEEE802.1Qci to assign a limited bandwidth and low priority to that entry point [13].

2.1.1.5 IEEE 802.1Qbv-2016: Scheduled Traffic

Until the introduction of this protocol, it was implausible to guarantee bounded end-to-end latency of time-sensitive data traffic, IEEE 802.1Qbv marked a significant step towards that goal. A packet with a low priority that is already transmitting could delay the transmission of a higher priority packet. IEEE802.1Qbv concept is to have specific time slots assigned to the transmission of time-sensitive data. During these time slots, it is guaranteed that no other packets or underlying Ethernet traffic will interfere with the transmission. This is of utmost importance for any kind of component that is safety-critical. The remaining will still be assigned a time slot for regular transmissions that may be interrupted, resulting in a much less deterministic behavior. Designers of the network need to do an analysis to know which functionalities require non-preemptable traffic [26].

2.2 AUTomotive Open System ARchitecture - AUTOSAR

AUTOSAR emerged as an open standardized architecture for automotive software agreed upon by the different OEM. It serves as a software layer to abstract from the underlying hardware and effectively transitioning from the system behavior and functionalities specified in the different IEEE standards to an actual programming interface. Following a layered software approach, starting from the highest abstraction level three different layers can be distinguished, the application layer as a composition of software components (SWCs), runtime environment (RTE) and basic software (BSW) [9] that can be seen in Figure 2.1.

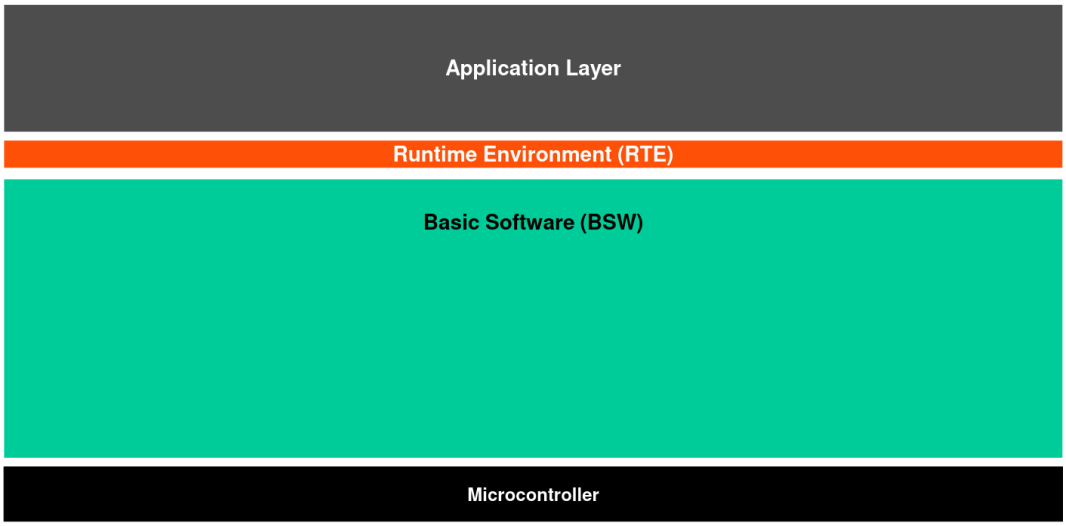


Figure 2.1: Overview of the 3 AUTOSAR layers (reproduced from [6])

2.2.1 Software Components

SWCs stand at the top layer and represent the different components of an application. These can be further divided into application SWC and sensor-actuator components. The difference being that the latter is ECU hardware dependent and is able to communicate directly with the ECU abstraction layer, the former is ECU independent and can be even further divided into different component types according to the needed functionality. Components can have private memory sections that provide private states to each. Additionally, components can belong to a partition, which represents a single functional unit of the system. Each component contains smaller units called Runnable that represent the smallest chunks of code that can be executed. The execution of a Runnable can only be triggered by the RTE, which is also responsible for the communication between them and accesses to the BSW modules.

2.2.2 Runtime Environment

This layer is at the heart of the abstraction that AUTOSAR provides. RTE is ECU and project-specific and is generated individually for each ECU. It consists of the realization for a specific ECU of the interfaces of the AUTOSAR Virtual Function Bus (VFB), the name of the mechanism that allows communication between SWCs, with the hardware of the system and standardized services such as reading or writing from non-volatile RAM. Virtual in the sense that it represents a connection between SWCs, regardless of whether the components are in the same ECU or in separate ECUs, therefore requiring transmission through vehicle buses. The other logical function of RTE is triggering the execution of Runnables. An event is at the base of this trigger. The most basic of the events is a Timing event that is periodically triggered. The remaining events are triggered as a result of a communication activity or operation mode switches [9].

Through the VFB, RTE supports two basic communications paradigms, Client-Server, providing function invocations, and Sender-Receiver, providing message passing. SWCs configure

their ports to provide one of the two paradigms. Each paradigm can be applied to intra-Partition, inter-Partition and inter-ECU. Intra-Partition further encompasses intra-task and inter-task, the former for components mapped to the same Operating System task whereas in the latter they are mapped to different tasks. Inter-Partition communication occurs between components belonging to different Partitions but located in the same ECU and therefore cross the memory isolation boundaries. Finally, inter-ECU communication occurs between different ECUs and involves transmission through vehicle buses, thus being inherently concurrent and unreliable.

2.2.2.1 Client-Server Communication

In client-server communication, the client initiates the communication, sending a request to the server that may contain a set of parameters. The server continuously waits for incoming requests from a client, once it gets one, it performs the requested service and then dispatches the response to the requesting client. A single component can be both a client and a server depending on the application. As the RTE serves as the middleman between this communication, the invocation of a server request is performed by the RTE. From a client perspective, this invocation can be synchronous, waiting for the server to complete, or asynchronous. It is not allowed for a client to initiate direct communication with the server, thus bypassing the RTE.

2.2.2.2 Sender-Receiver Communication

Sender-receiver communication consists of a single message sent from the sender to the receiver. Any reply that may originate from the receiver is then considered an entirely different sender-receiver communication, independent of the first one, where the sender and receiver roles are then inverted. There are four possible receive modes a receiver can be set to. *"Implicit data read access"*, that gives the receiver runnable access to a copy of the data that will remain unchanged throughout the execution of the receiver. This copy is not necessarily a unique copy for each receiver. *"Explicit data read access"*, allows the receiver runnable to execute a non-blocking API call that enables polling behavior. In this mode, the receiver needs to explicitly request the data in order to receive it. *"Wake up of wait point"*, where similarly to the explicit access it allows the receiver runnable to explicitly request the data. However, instead of a non-blocking API call, a blocking call is used and once the data is available, the call returns. It is possible to set a timeout to this blocking call in order to prevent infinite wait time. Lastly, *"activation of runnable entity"* where the receiver runnable is invoked automatically by the RTE upon arrival of new data. To then access the newly arrived data, the receiver needs to use either an implicit or explicit data read access.

2.2.3 Basic Software

Basic Software is the lowest AUTOSAR layer that provides services to the SWCs, it does not perform any functional requirement, but contains standardized services and components that can be used by the SWC through the RTE. In itself, BSW is separated in different layers, the Services,

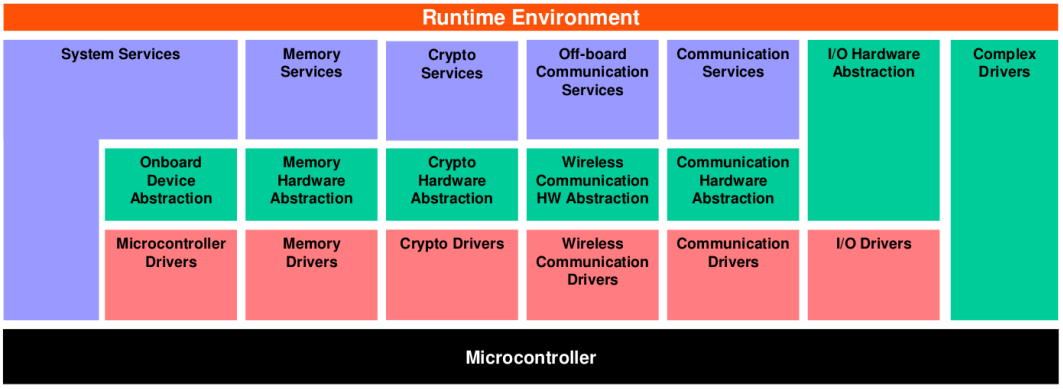


Figure 2.2: Components of the Basic Software layer, Services layer in blue, ECU Abstraction layer in green, Microcontroller Abstraction layer in red and the Complex Drivers also green (reproduced from [6])

ECU Abstraction, Microcontroller Abstraction and the Complex Drivers. In turn, these are also subdivided as can be seen in 2.2.

The Services layer is the highest layer of the BSW as it is also the most relevant for the application software. It provides, among other functionalities, operating system functionality, network communications, management services and memory services. While the rest of the Basic Software is hardware dependent, this layer is mostly hardware independent [6]. The ECU Abstraction layer, located above the Microcontroller Abstraction layer, abstracts from the specific ECU schematics. As such it is implemented for a specific ECU and is hardware dependent, it offers access to the different devices on the network regardless of their connection to the microcontroller, thus abstracting the layout of the hardware. Inside this layer, is located the Communication Abstraction that allows interfacing with different communication protocols, including CAN, Ethernet and FlexRay. The lowest layer of the BSW is the Microcontroller Abstraction layer where it is no longer the ECU that is abstracted, but the actual microcontroller that is present in the ECU. Hence it is microcontroller dependent and consists of drivers that map the access of on-chip peripherals devices of a microcontroller and off-chip memory-mapped peripheral devices to a defined API [45].

2.3 Service Oriented Architecture

Service-Oriented Architecture is a different paradigm from the older architectures where the system is divided in sub-systems and that were interdependent from one another. In a SOA, the different system resources are packaged into a single "service" that provides a specific system functionality while keeping their own internal state. The functionality of a service is specified in a standard definition language that will be shared among every element of the system. A component that implements a service represents a single instance of the service, and will then become a service provider, identified by a service instance. When a client wants to use a service instance, it simply needs to follow the definition language specifications to request the service. Once that happens, the service provider is now responsible for receiving the data and possibly reply to the

client. In the meantime, the client is free to continue its execution or wait for the reply. Note that it is possible for multiple instances of the same service to exist in a single system. In those cases, there will need to be a service discovery and instance selection process. While it would be possible to have the client handle this process, it would add complexity to the system. Hence, technologies that are developed with SOA in mind already provide this "service aggregator". The aggregator performs a dual role. It acts as a service provider to the service client and also acts as a service client to the actual different service providers [34]. The selection of which service instance to use falls to the requester.

This type of architecture is of great interest to any system with several components for a number of reasons. It ensures loosely coupled software modules as every service holds just the information required to execute its logic and regardless of the state of the system as a whole it will keep executing its functionality as long as the requests follow the definition language specification. Due to this fact, a SOA is platform and language agnostic. As long as the request or response follow the specification, it is unnecessary for either the client or the service to have insight into the implementation details of one another. Thus, the integration of heterogeneous components that may have been designed by different vendors is made seamless. Additionally, a SOA does not require a static system configuration. Through the use of service discovery, service providers can be discovered at runtime. This effectively means that it is possible to 'hot-plug' components into the system. No longer does the whole system need to go down in order to update or add a new component.

An already widespread application domain for SOA are Web services. The whole World Wide Web in itself is a SOA, users make requests to remote servers, with the specific HTTP format, and then the servers send the responses back to the requester. The automotive industry has also started incorporating a SOA in their products. With the vision of autonomous driving and enhanced driver assistance systems, not only do the existing automotive requirements apply such as functional safety and security, but also the ability to support High-Performance Computing, updates over the air and dynamic deployment of applications. An evaluation by AUTOSAR concluded that these requirements cannot be met by the classic architectures where nearly all internal vehicle communication is handled by a deeply embedded controller [15]. Hence the industry started looking into the applicability of existing SOA technologies in the automotive sector for these newest functionalities, while still maintaining the traditional system architecture for the most critical components.

2.4 GENIVI Alliance

In a vehicle, one of the most complex components, software-wise, is the head-unit, approximately 70% of the total code in a vehicle will in be in that single device [37]. It is no longer simply a radio and has evolved into a hub where passengers have control over the several systems of a car and has been named In-Vehicle Infotainment (IVI). The GENIVI Alliance, founded in 2009 by a group of automakers [46], is an alliance specifically formed to create a standardized development

platform to design IVI systems using Ethernet as its backbone. Unlike AUTOSAR, which simply provides standards, leaving the actual implementation of these to the automakers, resulting in each having their own AUTOSAR compliant product, GENIVI aims to introduce the concept of open-source development to the automotive software industry. All of its solutions are available for commercial use and open for everyone to use and make modifications. With this approach, that results in automakers cooperation, it also desires to increase the product innovation rate. GENIVI, however, did not intend to be a wholly separate alliance with no ties to other successful automotive alliances, for example, AUTOSAR. It was still relevant for its solutions to be AUTOSAR compliant, which was already widely used in the automotive industry. Consequently, one of its founding partners, BMW, that belongs to both the GENIVI and AUTOSAR alliances and thus already has AUTOSAR compliant modules, was able and willing to make these modules available to the remaining members of the GENIVI alliance. Then these members would be able to integrate the modules into their GENIVI solutions. The aggregation of these solutions is called the GENIVI Platform, and only around 20% of its software was created or adapted by GENIVI, the remaining 80% are adopted from existing open source technologies [37] that are needed for each particular system.

A necessity that arose was the need for a middleware solution able to provide a hardware and software abstraction to meet the automotive requirements whilst providing the services upon which the applications depend, and still be compatible with AUTOSAR and other alliances. The middleware would allow for a standardized way to not only communicate between applications in the GENIVI platform, but also applications in other platforms. These specifications fit a SOA perfectly, and thus GENIVI searched for a fitting existing middleware that could be deployed on the automotive market. The results were that although several middlewares already existed, out of them, only a few are suitable for an embedded environment, and none of these was compatible with Ethernet and AUTOSAR compliant [42]. As a result, the GENIVI alliance took up the challenge of developing a new middleware fit for the automotive market, and thus SOME/IP appeared.

2.4.1 Service Oriented MiddlewarE over IP (SOME/IP)

Service-Oriented Middleware over IP is a message based middleware designed in 2011 specifically for the automotive industry. It can fit on devices of different sizes and operating systems, like cameras, head units and AUTOSAR devices. The original use cases for SOME/IP were related to infotainment systems, although it is fit for other vehicle domains allowing it to replace some of the traditional vehicle buses [43]. As a middleware designed with SOA in mind, it is based on the creation of applications that implement a service definition and then offer it to the network, this is made easy by the range of features that SOME/IP provides. The middleware handles the serialization and deserialization of messages, and the message on-wire format is compatible with AUTOSAR control messages. A request-response communication is supported through the invocation of Remote Procedure Calls (RPC) as well as a publish-subscribe paradigm with the possibility to subscribe to remote events. Fields are also a possibility, having features of both request-response and subscribe-publish. They are a combination of one or more of the following.

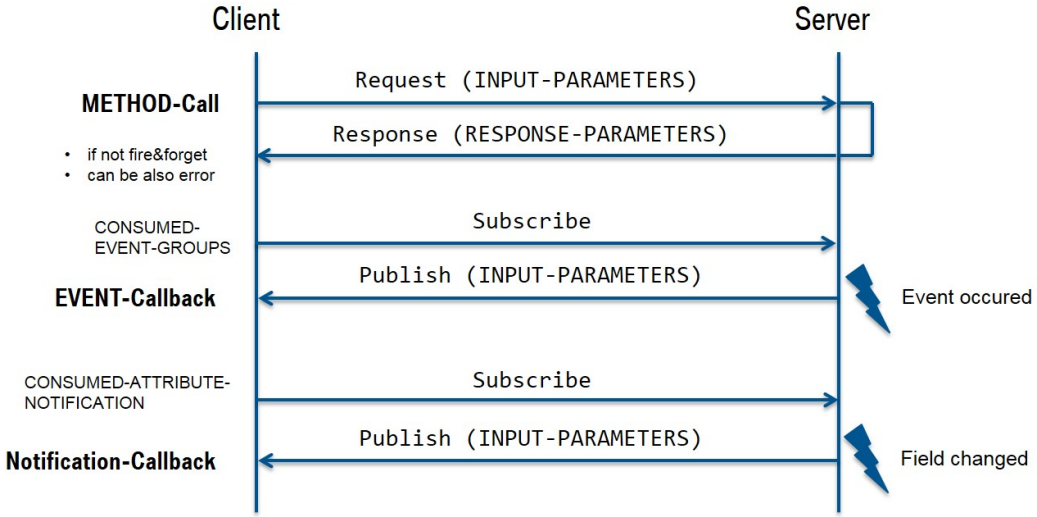


Figure 2.3: Overview of SOME/IP communication mechanisms (reproduced from [16])

A notifier that sends data to the subscribers when it deems appropriate, a getter function called by the client to fetch the current value of the field, and a setter function called by the client to change the value of the field. This is different from the events behavior that are only sent on change.

At the core of SOME/IP design is the Internet Protocol suite, also known as TCP/IP, a conceptual model and set of communications protocols used in computer networks. It is an already proven model for computer networks and is specially useful for Ethernet networks, one of the aims of SOME/IP. TCP/IP offers different transport protocols, the ones used by SOME/IP are Transmission Control Protocol (TCP) and User Datagram Protocol (UDP). UDP is a very simple protocol, offering only the base guarantees of multiplexing and error detection using a checksum, it does not guarantee that a message is received. As such, it is used for features that have a very short deadline and need to react quickly, as UDP lack of guarantees means messages arrive faster than TCP and with a predictable delay. On the other hand, TCP adds additional features for achieving reliable communication, not only does it handle bit errors, but also segmentation, loss, duplication, reordering and network congestion [7] at the expense of unpredictable delays. All of SOME/IP concepts allow the designer to select whether to use TCP or UDP, either statically or at runtime.

The discovery module of SOME/IP used to discover other available services on the network is called Service Discovery (SOME/IP-SD). It is optional and can be turned-off in case a static configuration of the network exists. If it is turned on, then every application will announce its existence to the network through the use of multicast messages. These are UDP messages that adhere to a different routing scheme. A one-to-one scheme, where a sender sends a message to the receiver, is called a unicast message. A multicast message consists of the sender sending a message to a multicast group that will be received by all subscribers of that group. This is opposed to broadcast messages that reach every device of the network. An application offering a service will keep sending these multicast messages, and a client application will then be aware of the service existence and can then proceed to make use of the offered services.

Apart from the standard SOME/IP messages, as this middleware is to be deployed as a complement to other existing vehicle buses, it can also serve as a tunnel for these vehicle buses frames. Support for CAN and FlexRay frames is possible, allowing SOME/IP to propagate these frames through the usage of a special identifier to distinguish from the regular SOME/IP messages. With this, the groundwork for the integration of SOME/IP with existing technologies was laid out. GENIVI aim, however, is to be as receptive as possible to newest technology trends and not being tied down to a single middleware or technology. Consequently, the CommonAPI was then developed.

2.4.2 CommonAPI

CommonAPI is a C++ framework developed by GENIVI for interprocess and network communication that follows a SOA. Its objective is to provide a uniform Application Interface (API) to different communication frameworks or protocols. The basis for a project using CommonAPI is the existence of a service definition, shared across the developers. To this end, the CommonAPI uses the Franca Interface Definition Language (Franca IDL). This language allows the definition of services with methods, events and fields, and also the specification of project-specific data types, such as structs or enums that will then have a specific on-wire representation. This is analogous to the specification of the message formats present in a SOA as it contains all the information a service or client need to encode and decipher messages. From this Franca file, CommonAPI provides code generators, the base one called `core-tools`, to generate code for both the service providers, called `Skeletons`, and consumers, called `Proxies`. The former contains the default implementation of the specified methods, while the latter has methods that internally handle the call to the service provider and the whole networking process. A developer will then extend the generated code, through C++ inheritance, without actually modifying the code files that were generated. This generated code uses methods from the CommonAPI Runtime, which is the actual abstraction layer from the underlying communications protocol. The base runtime is called `core-runtime` and has the higher-level concepts of a SOA such as remote methods, events and fields. Both of these core projects represent the API with which a programmer will have to interact when developing using this framework. A representation of this interaction is shown in Figure 2.4.

To then specify an actual middleware, additional steps need to be taken on the compiling process. The communication protocols that are supported until now are SOME/IP, D-Bus, a protocol for inter-process communication within the same system, and also Web Application Messaging Protocol (WAMP) which is still being developed [1]. For each of these protocols, there is a binding of the respective core project for that protocol, for example, the generator for code using SOME/IP is called `someip-tools`, and the runtime is called `someip-runtime`, the same logic applies for D-Bus and WAMP. In order to use one of these protocols, code for the respective protocol needs to be generated as well as the linkage, at compile-time, of the respective runtime to the application. Each of the protocol runtimes contains macros to dynamically link at runtime the respective binding to the core runtime, effectively providing networking capacity to it. Through this process, the developer is abstracted of the inner workings of the different communications

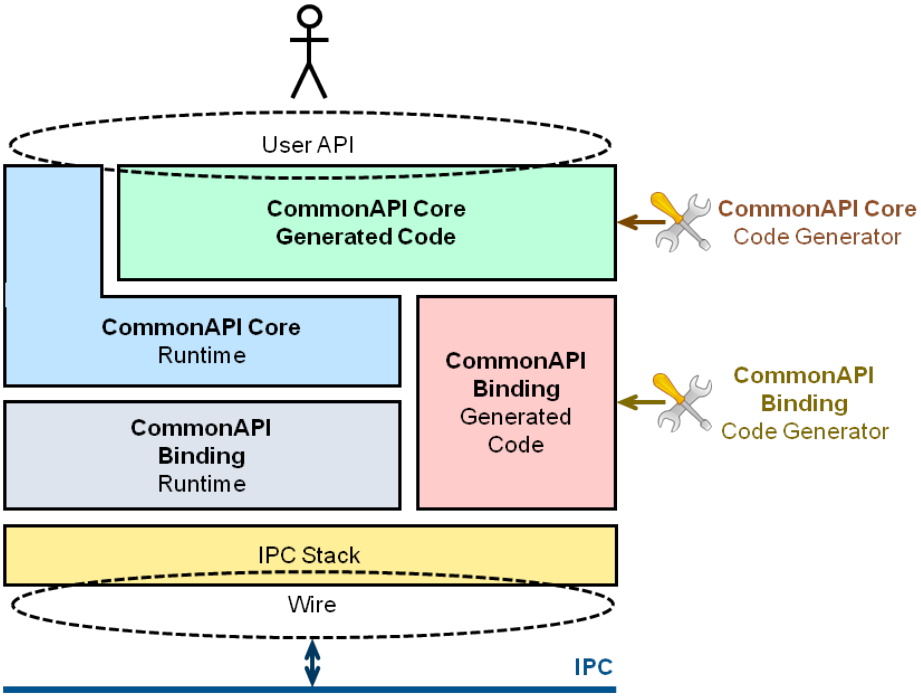


Figure 2.4: Overview of CommonAPI functional modules and relative abstraction level (reproduced from [17])

protocol and can only focus on the familiar services that they provide. This shift in the developer focus also makes it very easy to adopt new technologies. Once a project has been developed using CommonAPI, changing to another technology simply involves recompilation without even having to change the project. Also, as this follows the SOA and each Franca file will result in the generation of the messaging scheme and services provided, it is simple to have concurrent development, where a team is responsible for the service provider system and another for the service clients. As the generated code will be equal for the same Franca file, then focus is shifted from the networking part of the project to its actual functional requirements.

2.5 AUTOSAR Adaptive Platform

Soon after the appearance of other more specialized automotive standards and technology, AUTOSAR realized that while it does cover several domains, it is not reasonable to expect it to cover every single one. It seemed more reasonable to open AUTOSAR for an integration with these different platforms. Hence, in 2014, the AUTOSAR Classic Platform (CP) introduced support for SOME/IP [10] and also a Franca connector for supporting the integration of GENIVI and AUTOSAR at an application level and thus making it possible to interconnect the development and generation processes of the AUTOSAR and the GENIVI parts of the system [5].

However, as the CP could no longer meet the computing requirements of the newest automotive systems, in 2017 AUTOSAR introduced the Adaptive Platform (AP) as an additional compli-

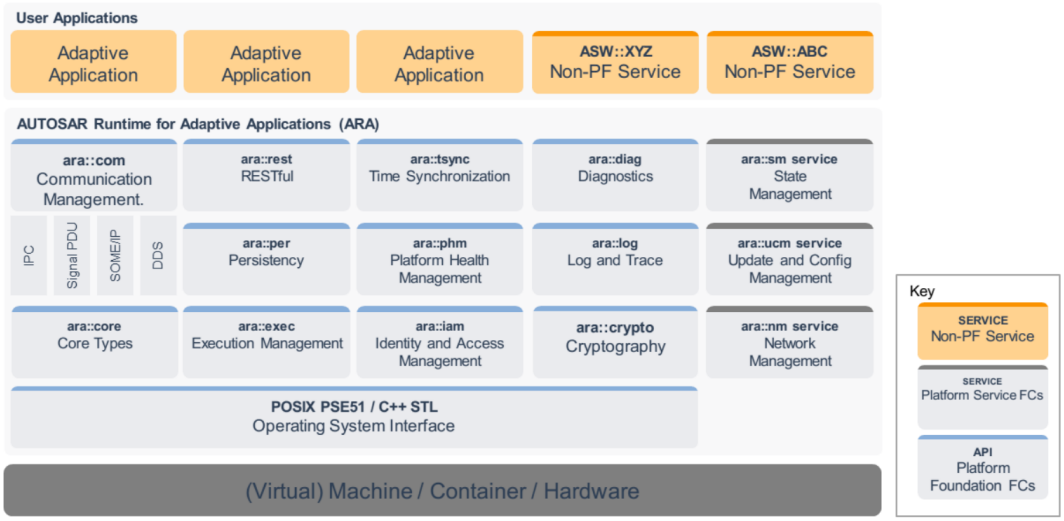


Figure 2.5: Overview of AUTOSAR Adaptive Platform functional clusters (reproduced from [3])

mentary standard providing more flexibility to the development of automotive software. Unlike the CP, which was developed for deeply embedded ECUs where their functionality does not change during the vehicle lifetime, the AP was designed with the dynamic nature of the state-of-the-art technologies in mind and their ever-increasing performance and bandwidth requirements. Not only multicore processors need to be supported, which was already the case with the CP, but also manycore processors and General Purpose use of Graphical Processing Units (GPU) that offer orders of magnitude higher performance than conventional processing units and overwhelms the CP. To tackle these requirements, AP employs technologies that were not used by the CP, while maintaining a degree of freedom in the implementation to allow the integration of innovative technologies. It can fully utilize the capabilities of Ethernet, allowing for much higher transmission rates than other vehicle buses. The CP already had support for Ethernet but primarily for the communication with legacy systems and was optimized for such, making it difficult to fully utilize its potential. C++ became the programming language of choice when tackling these issues, instead of the traditional C, which should provide faster adaptations of the newest algorithms while improving application development productivity if employed correctly. AP was designed following a SOA to allow the dynamic configuration of a system, achieving maximum flexibility and scalability. Thus, with a SOA, the system inherently fits the distributed computing paradigm where, through message passing, a network does computations towards the same goal. This message-based communication also greatly benefits from the Ethernet speeds. Lastly, to ensure the safety and security of the systems using AP, it employs dedicated functionalities and C++ guidelines that facilitates the safe and secure usage of such a complex language. Ultimately this translated into a platform that was not meant to be a replacement of existing ones but as a complement that will interact with other platforms, including non-AUTOSAR platforms [3].

Similarly to the CP, AP is divided into several functional clusters, presented in Figure 2.5. The whole set of functionalities is called the AUTOSAR Runtime for Adaptive Applications (ARA).

Each cluster belongs either to the Adaptive Platform Foundation, shown with a blue line, or the Adaptive Platform Services, shown with a grey line. The former provides fundamental functionalities of AP, while the latter provides standard services. In addition, any adaptive application can provide services to other applications, illustrated as Non-PF Services. Of these clusters, the one responsible for the communication between user applications is the Communication Management (ara::com) that provides abstractions comparable to GENIVI CommonAPI.

2.5.1 Communication Management

Similarly to the CommonAPI, the ara::com purpose is to abstract the developer from the mechanisms to find and connect applications. It also follows a SOA allowing the offering and consumption of services between applications that can be either in the same device or in different devices. These services consist of methods, events and fields, and its discovery can be established statically or dynamically through the use of Service Discovery. Correspondingly, code generation based on a service specification file is also standardized. Based on the AUTOSAR meta models, code is generated for both the service providers *Skeletons* and the service consumers *Proxies* that can then be extended by the developer. The actual communication protocol is abstracted. An application only interacts with ara::com API, which will then handle the interaction with the actual protocol. This approach is similar to GENIVI CommonAPI approach, but AUTOSAR also wanted additional capabilities not provided by CommonAPI or any other existing platform. Namely, receiver-side caches, already present in the CP, a zero-copy capable API with the possibility to shift memory management to the middleware and data reception filtering [4]. Therefore AUTOSAR designed a communication middleware API, based on existing ones, in order to meet its requirements.

The main protocol on which ara::com built on is SOME/IP. As it had already been designed for automotive purposes, it fitted the AUTOSAR requirements. It became the first protocol to be supported by ara::com, and it is an AUTOSAR requirement that any developed AUTOSAR AP compliant solution has to support SOME/IP, although the actual choice of the underlying middleware falls on the application developer. Another studied protocol was the Data Distribution Service (DDS), an Object Management Group middleware standard that builds on the concept of a "global data space" accessible to all interested applications [28]. DDS has been applied in different safety-critical systems, such as air-traffic control, transportation, and medical systems [21]. At the time of the creation of ara::com, DDS strictly followed a publish-subscribe communication mechanism, with no support for request-response. Hence it did not meet the required criteria to become a part of ara::com possible middlewares. Shortly after ara::com release, DDS introduced the possibility of RPC in its model [27] and some months later AUTOSAR added DDS binding to ara::com [8] adding DDS to the possible middlewares through which applications communicate.

In the end, the AP ended up with a workflow that is similar to GENIVI. As such, when using the same communications protocol, interoperability between AUTOSAR and GENIVI or other platforms comes naturally. The platform context switch is done one layer above the middleware, in AUTOSAR AP case with the ara::com and in GENIVI case with the CommonAPI. This opens up the possibility of integrating even more platforms in a vehicle or even using AUTOSAR for

non-automotive purposes, such as the Robot Operating System which is a widely used platform to develop robots, drones and other cyber-physical systems [35]. In this case, the protocol that allows the interoperability is DDS.

2.6 Rust Language

Rust is a programming language originally designed at Mozilla Research while developing Mozilla Firefox newest browser engine Servo. It then became an open-sourced language, thus encouraging contributions from throughout the world. Ever since it first appeared in 2010, its popularity and adoption have been growing tremendously, and it has been the "most loved programming language", according to the Stack Overflow Developer Survey, for five consecutive years [29, 30, 31, 32, 33]. What makes Rust so unique is its focus on providing a code safety level whilst maintaining a speed on par with C and C++ [39].

The solutions and platforms explored before, GENIVI and AUTOSAR, were developed in either C or C++. The issue with these languages is that the programmer is given complete freedom. They assume that as long as the code is syntactically correct, that it is valid, even in cases where it evidently is not and will result in an error. This approach would be perfect in case the developers had absolute knowledge over which approaches are the correct ones and the inner workings of the languages. However, it is not feasible to assume this as the amount of effort and time it would take to become fully proficient in these languages is massive. Solutions and standards to mitigate this have been proposed and currently exist. One of such standards is the MISRA-C and MISRA-C++ guidelines, a set of coding rules that, when followed, mitigate to a certain extent the likelihood of errors related to misuse of C/C++. These are simply guidelines, actually checking code against these guidelines is left to the developer side. To facilitate this process, there exist tools, some free to use others sold as a product, to programmatically check the code. As one can imagine, the workflow of developing against these various guidelines is immense and takes away time from the actual development.

On the other hand, Rust provides all of the analysis explained above at compile-time, effectively merging the workflow into a single step. At the heart of Rust analysis is a strongly typed language and its ownership model that is able to guarantee memory and thread-safety. The main ownership model rules are that every variable or value has an owner, owners can be blocks of code, such as a function, or data structures. There can only exist a single owner to every variable, and once the owner goes out of scope, the variable value will be dropped. Ownership can additionally be transferred throughout the code, for example, a setter function will first take ownership of the value and then move the ownership to the variable we are trying to set. By having these three simple rules, Rust is able to avoid having a garbage collector and makes it predictable when memory will be freed.

Complementing the ownership rules are the borrowing rules. These allow another piece of code to *borrow* the ownership of the variable. The borrows obey lexical scope that ensures there will not be any outstanding borrowed references to the object, also known as "dangling pointers"

[25]. Borrows can be immutable, where the borrower can only read memory, or mutable, where the borrower can read and write to memory. Mutable borrows have a uniqueness property. There can only exist a single mutable borrow and no immutable borrows to a variable in that scope. The owner of the variable is also forbidden from modifying the value while it is mutably borrowed. On the other hand, there is no restriction to the number of immutable borrows of a variable, but again, there can exist none once a mutable borrow is desired. When these borrows traverse the variable code scope, such as borrowing a variable to a function, then additional *lifetime* parameters need to be specified which allow the Rust compiler to verify if the borrow will remain valid throughout the execution of the function. Rust does provide a convention where these lifetimes can be elided on a general basis, thus remaining transparent for the developer [44], but the concept still remains.

To extend these concepts to a multithreaded context, Rust has two additional markers that represent when a type is thread-safe. The *Send* marker is used to represent that a type is safe to send to another thread. By sending a type to another thread, ownership is effectively being changed from the current thread to another thread. *Sync* marker builds on top of the *Send* and represents that a type can be safely shared between threads, usually a *Sync* type is also *Send*. This commonly involves the use of inner locking mechanisms, such as Resource Acquisition Is Initialization (RAII) guards or atomic variables. These markers are inherently given by the compiler to types that fit the criteria. A data type that is user-built, for example, a struct, will only be *Send* if all of its members are also *Send*, the same applies for *Sync*.

While these concepts are great and most projects will have no issue with following these rules, it is possible that a subset of features requires the violation of these rules. In such cases, Rust provides *unsafe* code blocks, where these strict ownership and mutability rules are relaxed. Inside *unsafe* blocks, Rust effectively becomes similar to C/C++ where it trusts that what the developer is doing is safe. As can be expected, most of the Rust compiler itself is *unsafe* and provides safe abstractions to the developer. The use of *unsafe* in Rust is not discouraged in itself but is the last resort concept that only after exploring all the safe alternatives should be considered. Since the compiler rules are relaxed in *unsafe* code, if there is any kind of memory safety error in the program, it is very likely for it to be in these blocks, making it much easier to actually find and correct the error.

Ultimately, these rules enforced by the Rust compiler ensure that a program will have no aliasing, data races or double frees. In [36] it was shown that several of MISRA-C rules are already enforced at compile time by Rust compiler, thus making such guidelines irrelevant. Most of the errors associated with memory safety are effectively eliminated as the compiler will catch them and fail to compile the code due to it, allowing the developer to fix these errors. Many errors present in a C/C++ project are precisely memory safety errors, with Microsoft estimating around 70% of its Common Vulnerabilities and Exposures are memory-related [40]. Due to this, it is looking to use safer languages and concepts and is exploring the possibility to implement Rust components in the Microsoft environment [41]. This follows the trend of other enterprises and projects that have begun to slowly adopt Rust. Discord, a proprietary Voice over IP application, has recently made a full transition from Go to Rust in order to meet their performance targets [18]. Facebook, one of the

biggest social platforms with a huge code base will integrate components written in Rust in their product [14]. Amazon also added support for Rust in its Amazon Web Services Lambda platform, which allows developers to run code for virtually any type of application or backend service [11]. There is clearly an industry trend to explore the possibility of using Rust in its products with clear advantages of doing so. Further efforts on developing kernels fully in Rust have been made, such as Redox, a Unix-like Operating System (OS) [12]. Using Rust for embedded purposes is also being explored, with Tock OS providing a very exciting full Rust implementation of an embedded kernel [22] [23].

2.7 Existing SOME/IP implementations

Apart from the publicly available GENIVI SOME/IP implementation, to the best of the author's knowledge, there is no other SOME/IP implementation nearly as complete. It is, however, possible that car manufacturers or their suppliers have their own implementations of the protocol only used internally and not shared with the industry. Only 2 other implementations could be found. The first one, [1] is also a C++ implementation of SOME/IP. However, this implementation no longer seems to be maintained as the last change to the protocol logic was made in June of 2015. As such, it is likely that the implementation has fallen behind the SOME/IP specification and is no longer compliant to it. The other implementation was made in Python [2] and appears to be a more complete implementation. It only covers the SOME/IP layer and not the CommonAPI, so the serialization and deserialization of data types is left up to the user application. Regardless, there appear to be some features missing when compared to GENIVI implementation as can be seen in the `README.md` of the repository. Namely, SubscribeAck and SubscribeNack messages are ignored, which means that user applications cannot react to their remote event subscription being either accepted or rejected. Additionally, it is not possible to subscribe to events that are offered in TCP among some other features. While this project is in active development, and its certainly interesting to have an implementation in Python, it seems it is not yet possible to develop a fully fledged user application with it.

2.8 Summary

In this chapter, an analysis of the process of the adoption of Ethernet connectivity within an automobile was given. At the base of this adoption was the IEEE effort to establish an aggregation of standards and respective configurations that fit the automotive industry profile. This served as the groundwork for the development of technologies based on Ethernet. As a worldwide accepted standard, AUTOSAR added support for it, initially in its Classic Platform. With the usage of Ethernet for connectivity, the additional usage of the TCP/IP protocols was made simpler. Other software industries, already using Ethernet and TCP/IP and with a substantial code base, began

[1] jacky309 someip repository
[2] afflux pysomeip repository

moving to a Service Oriented Architecture as an answer to the increasing complexity in the development of software. The automotive industry soon started following this type of architecture. As such, BMW, as part of the GENIVI Alliance, introduced a new protocol, SOME/IP, based on a SOA, specifically designed for automotive use cases. Later, AUTOSAR also adopted SOME/IP first in its Classic Platform and then, once it emerged, in its Adaptive Platform that was devised with a SOA in mind. By having this common communications protocol, both of these platforms are easily interoperable as well as any other platforms using a shared communication protocol. Most of these protocols and standards target either C or C++. However, these are, by nature, unsafe languages that rely on the programmer making proper usage of their concepts. Mozilla created an additional programming language, named Rust, that provides concepts similar to C and C++ but offering a much higher code safety level by having a powerful compile-time code analysis. Due to this, parts of the software industry have already started exploring the possibility of using Rust in its products. Additionally, there is some research and development into using Rust for the embedded software world. The automotive software industry, being a technology that needs to be safe and robust could benefit from the usage of Rust, hence the proposition of this thesis.

Chapter 3

Reference toolchain analysis

To better analyze the capabilities of the presented technologies, a simple system example will be used. Naturally, it is not a system representing an actual use case, but one that utilizes the major SOME/IP and CommonAPI functionalities. It will be composed of 4 different services that could hypothetically be present in a car, as shown in Figure 3.1.

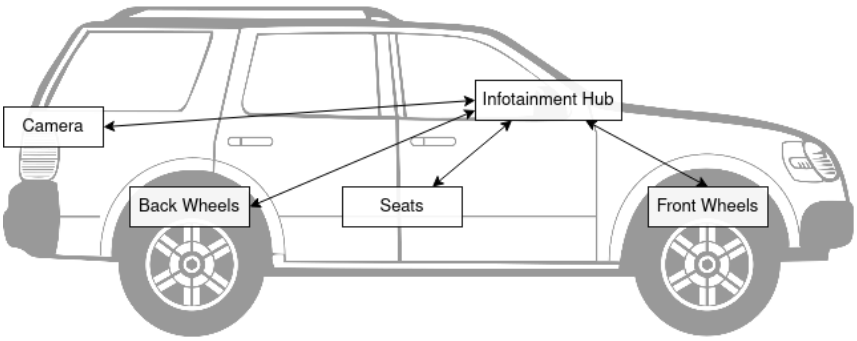


Figure 3.1: Overview of the example system with its services and how they communicate between each other

The first is the Infotainment Hub representing the head unit component, which is regularly used both to display vehicle information to the driver and also allow the driver to change the behavior of other vehicle components. The second service is the Seats, responsible for all vehicle seats related information. Similar to actual automobiles, it gathers the status of whether the various seat belts are locked and manages the heating of the different seats that will be further subdivided in bottom heating and back heating. The third service is the Camera component that would be located at the back of the vehicle. Its functionality is equivalent to that of a parking camera, and it is only activated once the vehicle goes into reverse mode, otherwise being turned off. Lastly, the Wheels service corresponds to a component responsible for each wheel axle of the vehicle. As such, there will be 2 instances of this service, one for the front wheels and another one for

the back wheels. Each of them will oversee their respective pair of wheels, gathering information about their status, such as their pressure and whether there has been any kind of damage to the wheels. The control of the vehicle's speed is left to an external component not presented in this system, the Wheels service only reads the actual speed as measured by the sensors on the wheels. As opposed to the more complex network structure of an actual vehicle, all of the services in this example system will be connected to the same network. However, only the Infotainment Hub will communicate with every other component. The remaining only need to be aware of its existence.

To simulate the automobile's movement and thus trigger certain events, a formula will be used to calculate the vehicle's speed throughout the simulation. Each of the Wheels services will use the following formula in order to generate speed variations.

$$7\cos(0.3t)+5$$

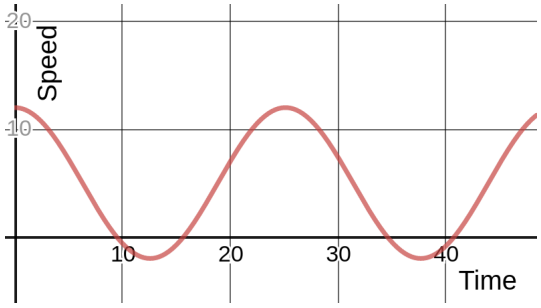


Figure 3.2: Plotted graph of the vehicle's speed variation through time

3.1 Project development workflow

The modeling of this example system in the CommonAPI toolchain needs to follow a particular workflow. After having the conceptual architecture of the system, it needs to be realized in the necessary format. With the CommonAPI, the initial-step is to define the system services in Franca Interface Definition Language (Franca IDL) files. These will have a description of services, alongside their methods and events, independent of the technologies that will be used to develop the system, thus effectively following a Service Oriented Architecture. The first type of files, and the ones that contain the descriptions of the services, are the `.fidl` files. These will have all the necessary information to describe the services, including the methods and events data types. The data types can either be the ubiquitous primitive types, such as numbers or strings, complex types like arrays or maps, or user-defined types, such as enumerations or structures. For the latter, they also need to be defined in the `.fidl` file, alongside the service methods and events. With the `.fidl` files, we get a technology stack independent service description. However, each specific technology has different ways of handling the services instances and how their intercommunication works. To then bind these service instances to a specific technology, additional deployment files, named `.fdepl`, need to be created, these will be technology-dependent. In the

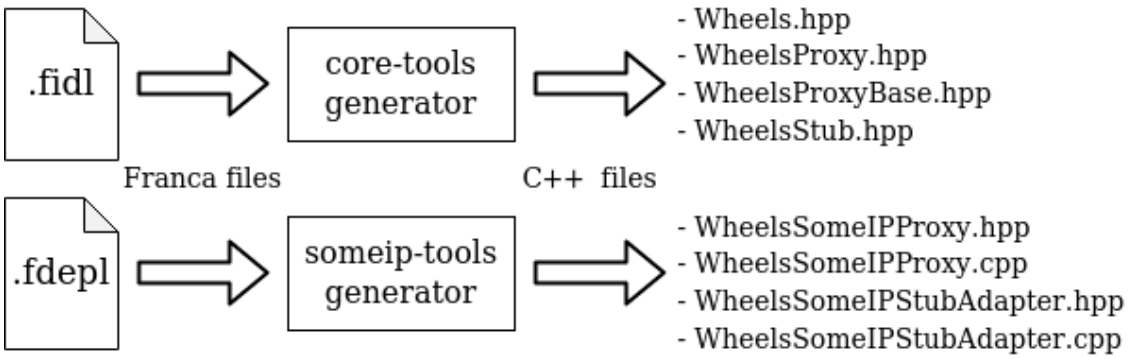


Figure 3.3: Transformation of the Franca files by the code generators to their C++ representation

case of a SOME/IP deployment file, a unique service identifier has to be manually assigned to each service and an identifier for each intended service instance. As per SOME/IP rules, the pair service identifier and instance identifier needs to be unique. Then, additional identifiers have to be defined and assigned to the methods and events of the service, as well as the ability to choose the reliability of the method, which translates to either using TCP or UDP as the underlying transport protocol. Several other adjustments are possible, such as using little or big-endian and multiple customizations of the on-wire representation of the several data types. The files that were used for this example are present in Appendix B.1 and B.2.

3.1.1 Code generation

Once the desired `.fidl` and `.fdepl` files are finished, the code generation takes place. This step will generate code that matches the service specification in the Franca IDL files. Since this code is very similar between services and thus prone to repetition, instead of having the users write this code by themselves, the code generator does it. GENIVI CommonAPI provides multiple code generators with all of them generating C++ code but with different purposes. The primary code generator, named `capicxx-core-tool` (`core-tools`), accepts `.fidl` files as input and outputs code which is technology independent. It can also accept `.fdepl` files, but it will only process the associated `.fidl` files. The output consists of C++ abstract classes with unimplemented methods that have to be implemented partly by the user and partly by the code generated from the rest of the CommonAPI generators. Any user-defined type will also be translated into C++ code in the `core-tools` generator as its definition is also technology independent. The remainder of the code generators outputs a technology-dependent implementation of previously generated abstract classes. For each technology supported by CommonAPI there is an associated code generator, the SOME/IP one is named `capicxx-someip-tools` (`someip-tools`) and will be the sole focus of this work. An overview of the pipeline each Franca file goes through to be transformed into their respective C++ code is shown in Figure 3.3.

As can be seen, the `core-tools` generated files are all header files that only contain abstract classes. The `Wheels.hpp` contains some public basic general data access methods with no logic. The `Proxy.hpp` files represent the interface that will be used by the clients of the service. As such, it contains the definitions of the service methods and provides the means to access the service events. The `Stub.hpp` interface file has the skeleton that serves as the base for the service implementation. To provide this implementation, the user needs to extend the Stub interface in this file and write implementations for each of the methods there present. In the `someip-tools` side, the generated files will be providing a transformer between the high-level Franca IDL concepts and their SOME/IP equivalents. The `SomeIPProxy` files are responsible for the client part of the service and thus, initially serializes the user's parameters, sends them to the service instance and then deserializes the response from the service into the respective type. The `StubAdapter` does the opposite, initially deserializing the bytes sent by some client, sending the result to the user's `Stub` implementation and afterwards serializing the response and sending it back to the client.

There can be additional files generated, named the `Deployment` files, that are generated according to the `.fdepl` files specification, where the user can control how the on-wire representation of the data types is handled. However, these `Deployment` files do not have any logic in them, but only store this on-wire specification which will then be used by the logic in the remaining files.

3.1.2 CommonAPI Runtime

An essential part of the CommonAPI toolchain are the Runtimes. These are where the actual logic for the transformation between the high-level Franca concepts and their middleware equivalent takes place. The generated code serves as a mean to abstract the underlying calls to the Runtime from the user. Similar to the code generators, there are several CommonAPI Runtimes. The common one, named `capicxx-core-runtime` (`core-runtime`), is technology independent and mostly consists of abstract classes to be implemented by the remaining Runtimes. The remaining Runtimes are the technology-dependent implementations. The SOME/IP one is called `capicxx-someip-runtime` (`someip-runtime`).

All of the generated code will be built on top of the `core-runtime`, as it provides a middleware independent abstraction over the high-level Franca concepts, making switching between technologies straightforward. The `core-tools` generated code files will then only use the `core-runtime` types and the `someip-tools` files will internally contain `someip-runtime` which, since they are an implementation of the core abstract classes, allows for the application to remain technology independent. This is possible due to the polymorphic capabilities of C++ both at runtime, through dynamic dispatch, and at compile time using static dispatch.

3.1.3 User Application

Once the previous steps have been taken, the user is now ready to develop its application. The base for this is the implementation of the Stub classes for each service. The user is free to decide how this should be done, for example, the same C++ class can implement 2 different Stubs. In the

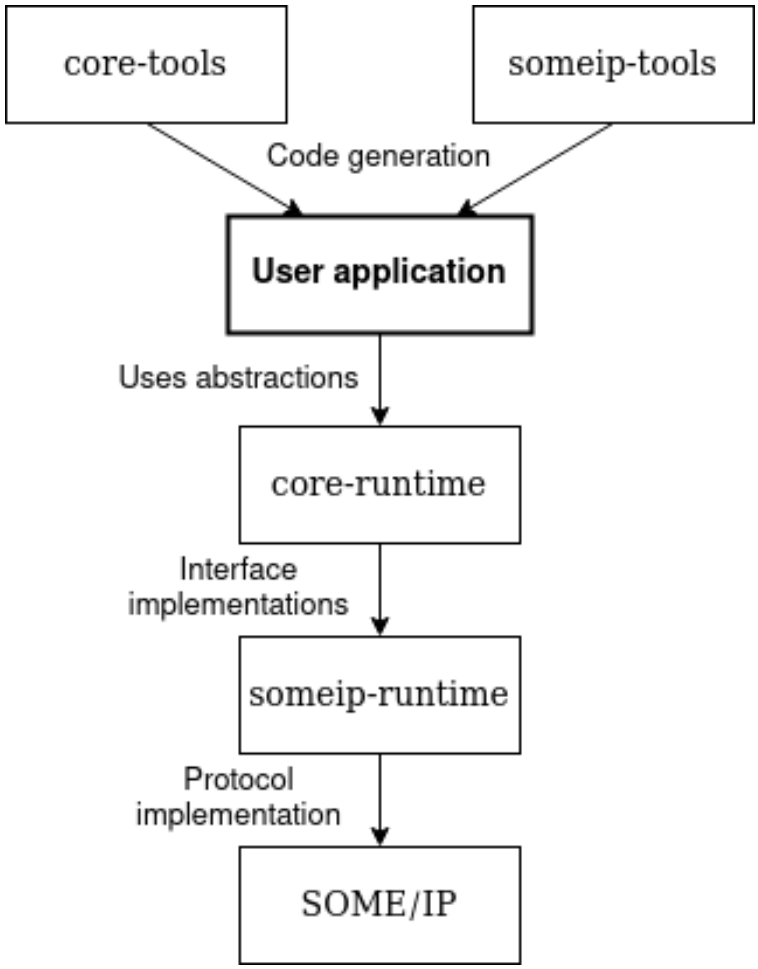


Figure 3.4: Overview of all the CommonAPI tools and how the user application uses its libraries

code generator, a flag can be used to generate default implementations of the service that the user can then reuse so that it is simpler to develop the application. As for the client part of the code, the user simply needs to instantiate Proxy class for the respective service and then it will be able to do remote method calls and access its events. How these Stubs and Proxies are handled by the CommonAPI will be detailed further in the document. From the user application point of view, it is only using a single library, `core-runtime`, although internally the library can dynamically choose which middleware implementation to use. An overview of all the CommonAPI tools and how each is used to allow the user to develop applications that are abstracted from the middleware is shown in Figure 3.4.

3.2 Service definition with Franca IDL

The Franca IDL allows for the definition of service interfaces using a variety of concepts. In the provided example, the most common and relevant aspects were covered. The most basilar aspect of a service specification is the definition of data types that will then be used on the remainder

of the concepts. Primitive data types are readily available and can be used without any further description, while the complex data types require the user to define them. The complex types encompass vectors, maps, enumerations, unions and structures. These can be associated with a specific interface or with a `typeCollection` and then be available for use in all interfaces. This visibility of data types only exists while creating the Franca files, naturally, this is not present in the resulting generated code.

A `typeCollection`, named AutomotiveTypes, was defined for the system to hold particular service independent data types. One such type is the `Pixel` structure, which stores the information of a single pixel of the images that will be sent from the Camera service to the Infotainment Hub. The standard Red Green Blue (RGB) representation was followed, with the additional alpha parameter used for mixing data types in the same structure. A single image from the Camera is called a `Frame`, which is a vector of `Pixels`, also defined in the AutomotiveTypes collection. For the data types associated with a specific service, they should be defined within an interface, an example of this is shown on the right side of Figure 3.5. In the Seats service, each car seat has dedicated heating. These have 2 different actuators, on the back of the seat and the bottom, so the `SeatHeating` enumeration stores which of these actuators should be turned on or off. Then, as the Seats service is responsible for the 5 seats in the vehicle, it additionally contains the `SeatInformation` map, where each of the 5 seats has an associated name, stored in the String, that then maps to the `Seat` structure which contains the heating and seatbelt information.

```
typeCollection AutomotiveTypes {
  version { major 1 minor 0 }

  array Frame of Pixel
  struct Pixel {
    UInt8 r
    UInt8 g
    UInt8 b
    Float a
  }
  ...
}
```

```
interface Seats {
  version { major 1 minor 0 }

  map SeatInformation {
    String to Seat
  }

  enumeration SeatHeating {
    OFF
    BOTTOM
    BACK
    BOTH
  }
  ...
}
```

Figure 3.5: Extract of a definition of data types. Left with types declared in a `typeCollection` and right with types declared in an interface

All of the Franca data types are then to be used in the remainder of concepts. For the specification of services, a ubiquitous approach is to define methods. In Franca, their definition follows the common standards to specify the input parameters, to be sent by the service clients, and the output parameters sent from the service provider back to the client as a response to the initial request, as seen in Figure 3.6. Theoretically, an arbitrary number of input and output parameters can be specified, however, programming language specific restrictions may limit this number. For exam-

```
interface Seats {
  version { major 1 minor 0 }

  method set_heating {
    in {
      String seat
      AutomotiveTypes.SeatHeating heating
    }
  }
  ...
}
```

Figure 3.6: Franca IDL definition of methods without any output or error arguments

ple, for the developed Rust implementation of these tools, a maximum of 12 output arguments can be specified, although grouping arguments in a separate structure can easily circumvent this limitation. Additionally, Franca provides the possibility to define a third type of parameter, the error parameter. This is a single parameter that is defined similarly to the input and output but is necessarily an enumeration. It will always be sent by the server, thus acting as an additional output argument that can provide additional information about the method call status.

For the developed system, not many methods are required as it is mostly a system that reacts to events without the need for more advanced querying. One of the few methods defined, `set_heating()` in the `Seats` service, allows the Infotainment Hub to update a single seat heating status. In it, the name of the specific seat is passed alongside the heating instruction. No output parameters were specified, but their definition follows the same principle as the input parameters, just using the `out` keyword instead of `in`. Although no output or error arguments where specified, the default Franca behavior is to send a reply with an empty payload that serves as an acknowledgment that the server has received and processed the request. This behavior can be turned off by adding the `fireAndForget` keyword in front of the method name, in which case, there is no reply sent by the service provider.

Lastly, one core concept of a SOA, which Franca follows, is the definition of events, shown in Figure 3.7. There are different types of events that can be used with Franca. The most basic ones are called attributes, and they represent a public service variable that service clients can access. This remote access can be restricted through a mixture of up to 3 keywords. To have a read-only attribute the keyword `readonly` should be used, for write-only `noRead` and no subscriptions `noSubscriptions`. These permission flags may be mixed, so it is possible to define an attribute that can only be read from with `readonly noSubscriptions`. Each attribute has an associated data type, though it is always possible to define a structure that holds different data types and then have a struct attribute.

The other types of events are called broadcasts, which get sent to every subscribed user. Unlike regular events, broadcasts cannot be read or written by the service clients. These can only subscribe to its notifications, and then the service provider is responsible for triggering it. Not only that, but a single broadcast notification can send multiple data types. From a user perspec-

```
interface Wheels {
  version { major 1 minor 0 }

  attribute Float speed readonly
  attribute Boolean sport_mode

  ...
}
```

```
interface InfotainmentHub {
  version { major 1 minor 0 }

  broadcast shutdown {
    out {
    }
  }
  ...
}
```

Figure 3.7: Extract of the definition of service events. Left with the declaration of attributes and right with a regular broadcast declaration

tive, broadcasts are used in a similar way to methods as it is simply a function call with no reply. There is a more restrictive type of broadcasts, called the selective broadcasts that can be defined by adding the `selective` keyword in front of the `broadcast` declaration. Unlike in the regular broadcasts where the notification will be sent to all connected clients, selective broadcasts may not be sent to all clients. The server is given the responsibility of selecting which of the clients to send the notification.

Most of the communication between services in the example system is through events. As an example, the Wheels service is only composed of attributes. Its attributes are either regular attributes or `readonly` attributes. The regular attributes are values that can be modified externally, the vehicle's sport mode can be activated in the cockpit, which will then write the value to the Wheels services. The other attributes represent values that are not dependent on the user's interaction with the system. Since we are assuming the car speed is determined by an external component, it is a `readonly` attribute for our example system. As the central hub for user interactions, the Infotainment Hub service has the `shutdown` broadcast. Every service of the system should subscribe to the broadcast on startup, and once the vehicle is turned off, a notification is sent to them in order to have a graceful shutdown. No output parameters where specified as there is no need, but as can be seen its specification follows the same principle as the definition of a method output parameters.

3.3 Mapping Franca concepts to the CommonAPI

The Franca IDL specifies several high-level concepts that need to be materialized into their concrete representation for the middleware being used. The code generator is part of this materialization. While it would be possible to generate code to match a specific middleware directly, GENIVI approach was to have an additional abstraction layer, the CommonAPI, to further abstract this generated code from the underlying middleware. For that, the base library `core-runtime` provides mostly interfaces that directly correlate to the Franca concepts, that need to be implemented separately for each middleware supported by the CommonAPI.

Figure 3.8 provides an overview of the main interfaces and abstract classes present in `core-runtime`, which are used to translate the Franca concepts or as helpers for a SOA. On the left,

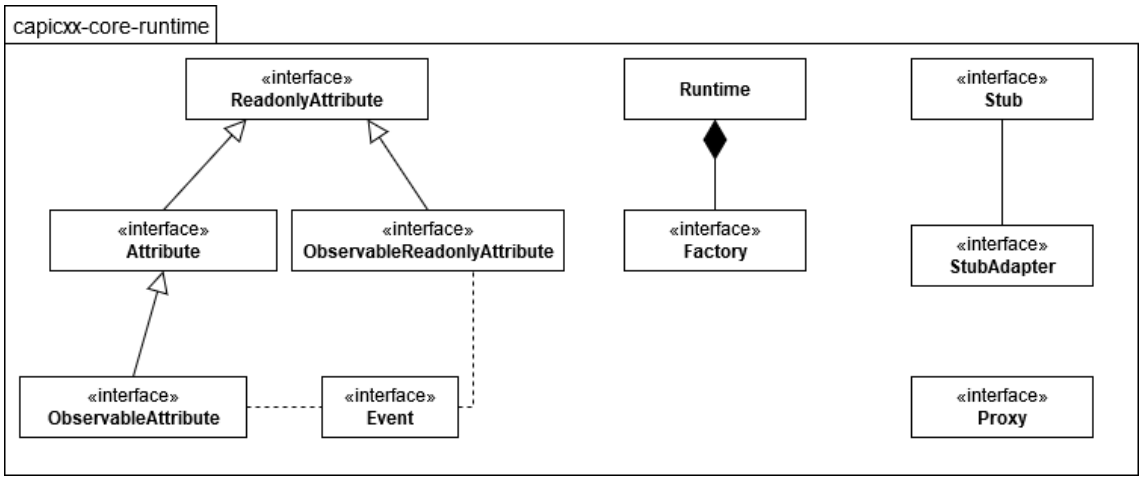


Figure 3.8: Diagram of the major classes and interfaces of the `capicxx-core-runtime` library

there are only event-related elements. There is an interface for each supported combination of the Franca attributes access modifiers, with the most restrictive attribute being the `Readonly-Attribute` that corresponds to the keywords `readonly noSubscriptions`. Attributes that can be subscribed to, are required to have access to an `Event` due to having the method `get ChangedEvent()`, which needs to return an event as shown in Figure 3.9. This `Event` access is required to manage the subscription and notification procedures of the attribute. The service broadcasts are mapped directly to an `Event` object as these cannot be written to or read from, only subscribed by remote clients. All of the event-related interfaces are generic over a number of template arguments that correspond to the respective data types of the Franca attribute or broadcast. As such, and as can be seen in Figure 3.9, the attribute interfaces take only a single template argument, while the `Event` is generic over an abstract number of template arguments, matching the broadcast capabilities of specifying an abstract number of output parameters.

On the other side of Figure 3.8, there is the `Proxy` and stub related abstract classes. This is the base for the code that will be used by the service clients and the service providers, respectively. The `StubAdapter` instance, to which the `Stub` object has access to, provides the translation from the underlying middleware bytes into the respective data types used by `Stub` methods and

```
template<typename... Arguments_>
class Event {
  ...
}

template<typename ValueType_>
class ReadonlyAttribute {
  virtual Event<ValueType>& getChangedEvent() = 0;
  ...
}
```

Figure 3.9: Extract of `capicxx-core-runtime` of the definition of the templated abstract classes for the `Event` and `ReadonlyAttribute`

events. Creation of the Proxies and registration of Stubs is done by the `Factory` component and each supported middleware will implement a `Factory` for this purpose. The only object that is used as is without any functionality extension by supported middlewares is the `Runtime`. It is a singleton class accessible everywhere and is used as the entry point for the CommonAPI. User applications will initially fetch the `Runtime` instance and then use it to create the Proxies and register Stubs. The `Runtime` will then dispatch this creation request to the respective `Factory` depending on which middleware the user wants to use. This is the core component that enables the CommonAPI to support multiple middlewares both at compile time and at runtime with the dynamic decision of which `Factory` to use.

3.3.1 Mapping to the SOME/IP Runtime

As the `core-runtime` mostly consists of interfaces and abstract classes with a high degree of modularity, it does not assume anything about how the Franca concepts are mapped onto the middleware. This approach allows the CommonAPI to, in theory, support many different communication protocols, as long as they follow a SOA, while remaining unchanged from a user perspective. Thus, each specific middleware will have its own implementation of the CommonAPI. We will only be looking at the SOME/IP implementation as that is the focus of this work.

An initial aspect that may seem odd is that the `someip-runtime` attributes do not have the inheritance in Figure 3.10. This is not entirely true, in fact, they do have an inheritance, however, it is not a straightforward inheritance. The `someip-runtime` attributes are generic over a template argument, that is required to be a `core-runtime` attribute, then they inherit from this templated argument as seen in Figure 3.11. There are indeed further template constraints inside the class, meaning that a SOME/IP `ObservableAttribute` can only accept a templated argument that contains either the core `ObservableAttribute` or the `ObservableReadonlyAttribute`. What this means is that in order to have a regular attribute with no access restrictions the type declaration would be

```
SomeIP::ObservableAttribute<SomeIP::Attribute<ObservableAttribute<...>>>
```

where the attributes prefixed by `SomeIP` are part of the `someip-runtime` and the attribute with no prefix part of `core-runtime`. In the `someip-runtime` there is now the extension of the `Event` abstract class, which is a member of the SOME/IP `ObservableAttribute` and is the object returned when the `getChangedEvent()` method, from `core-runtime`, is called. Note that in the SOME/IP implementation, an `Event` instance is never used by the service provider part of the generated code, so neither by `Stub` or `StubAdapter`. Its purpose is to serve as the entry point for the accesses of remote events, which it does so by owning an instance of `Proxy` and registering event handlers in it. This is because the remote accesses to a service event are, from a service provider point of view, mapped into methods. When a client wants to fetch an attribute value, it translates into a SOME/IP method call to the service getter method, with the same procedure happening for the update of an attribute value. The broadcasting of event notifications

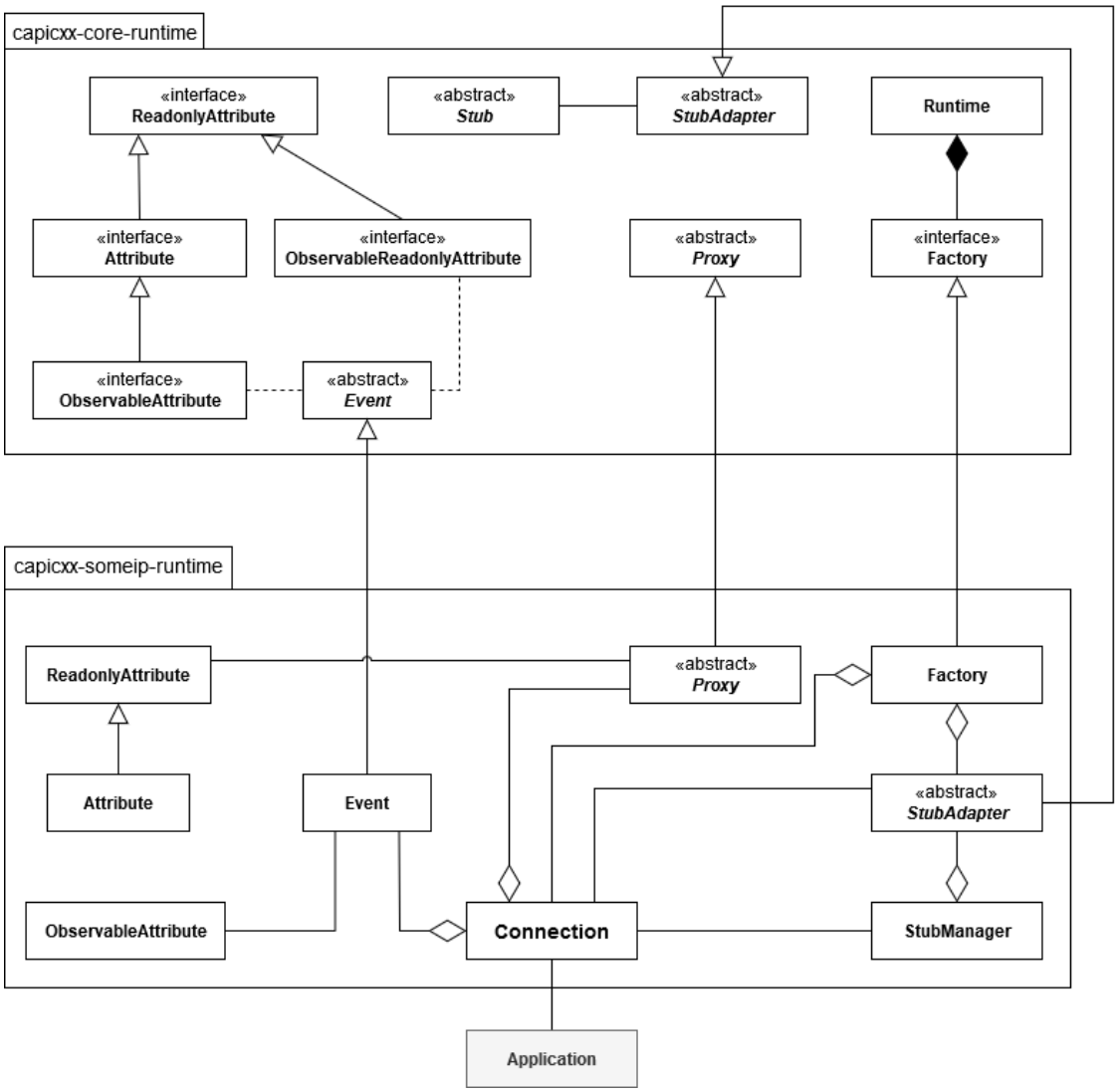


Figure 3.10: Diagram of the major classes of the `capicxx-someip-runtime` library and its relationship with `capicxx-core-runtime`

is also, from a user perspective, mapped into methods that will then use the vsomeip notifications API to send a notification message. In the end, this means that all of the event related classes and objects of the CommonAPI are only used by the `Proxies`.

On the right side of Figure 3.10, as expected, there is the SOME/IP implementation of `Proxy` and `StubAdapter`. The implementation of `Stub` is not made by each of the middlewares, but initially by the code generator that adds the declaration of the service methods, then ultimately by the user by providing the implementation of these methods. A new class that has been created is the `StubManager`. It stores all of the `StubAdapters` and associates them with their respective SOME/IP service instance pair. Once a SOME/IP message has been received, and in case it needs to be delivered to a `Stub`, it will be redirected to `StubManager` which will subsequently dispatch it to the respective `StubAdapter`. Then, the last implementation of `core-runtime`, `Factory`,

```
template <typename AttributeType_>
class ReadonlyAttribute: public AttributeType_ {
  ...
}
```

Figure 3.11: Extract of `capicxx-core-runtime` of the definition of the templated abstract class `ReadonlyAttribute`

which, as previously explained, is responsible for creating the `Proxies` and the registration of `Stubs`. While the `Proxies` creation is direct and involves no intermediate steps, the registration of `Stubs` is not. It firstly involves creating the `StubAdapter`, which itself receives and stores an instance of the `Stub`, then it sends an instance of itself to the `Stub`, so they both end up with a shared instance of one another. Once that process is completed, the `Factory` fetches the `StubManager` from `Connection` and registers the newly created `StubAdapter`, now with the respective `Stub` instance in it.

The final component, `Connection`, is where most of the `someip-runtime` logic is implemented and is the sole component that communicates directly with the SOME/IP implementation API. Its creation is done precisely by `Factory` during the creation of `Proxies` and registration of `Stubs`, and each may be associated with multiple `Proxies` or `Stubs`. `Connection` stores all the relevant information needed to manage all incoming SOME/IP messages directed to the service instances associated with its `Proxies` and `Stubs`. In the case of an incoming service request that needs to be redirected to a `Stub`, the `Connection` will send it to the `StubManager` which will then dispatch it to the respective `Stub`. Messages directed at `Proxies` can only happen as a response to a proxy request or as a notification for a subscribed event. In the former case, `Connection` can either do a blocking request, where the call blocks until a response has been received, up to a certain timeout, or a non-blocking request, in which case the `Proxy` registers a handler that will then be called once the request response has been received. When the message is a notification, it will be redirected to the respective `Event` object which will, in turn, execute all the subscription and notification handlers registered by the `Proxy`.

As mentioned, the fetching and updating of attributes is mapped into a SOME/IP method call so it is treated as a request response. Then, `Connection` also provides the possibility to register some reaction handlers, such as an availability handler, that gets called when the respective remote service of a `Proxy` becomes either available or unavailable, and also a subscription handler, called whenever there is a subscription state change to any of the subscribed events. These have a direct equivalent in the SOME/IP implementation API, and as such, they are simply redirected to it. The entry point for the SOME/IP implementation is the greyed out component `Application`, which is the component used for interacting with `vsomeip` and that will be analyzed in Section 3.4.

3.3.2 Control flow of a CommonAPI application

As could be seen, the CommonAPI library contains numerous components with plenty of associations between one another, making it easy to get overwhelmed. To better consolidate the

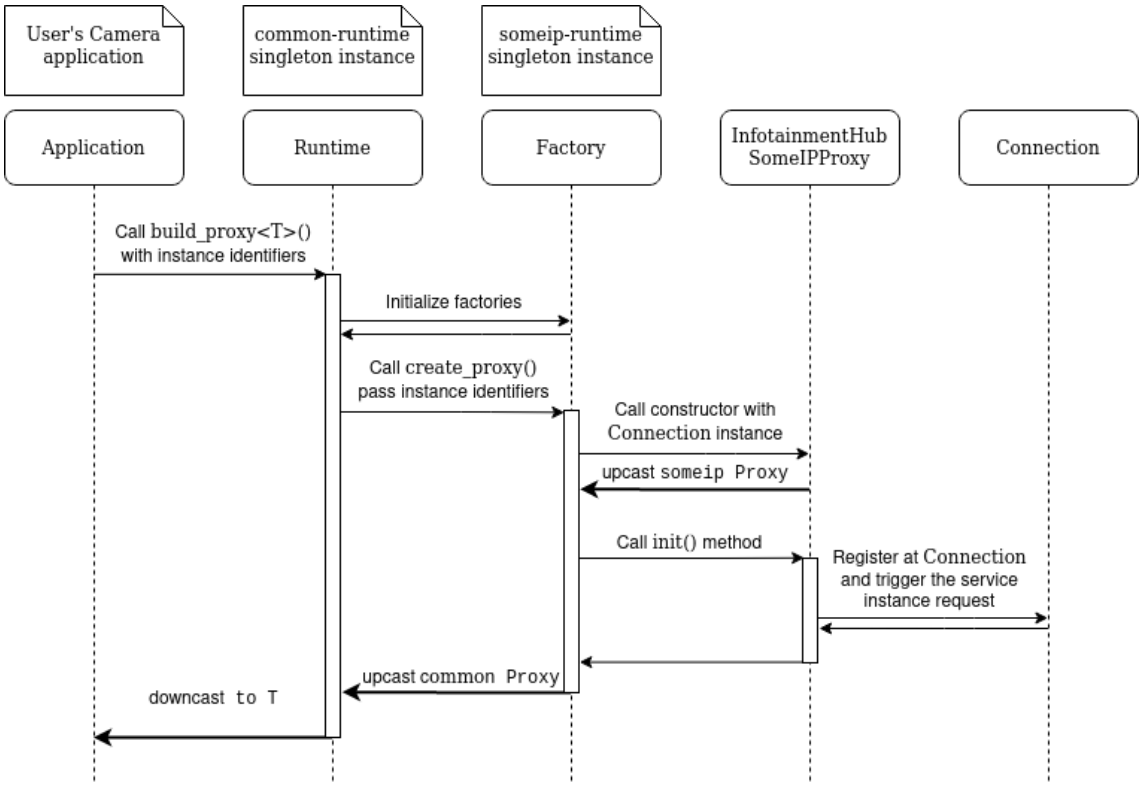


Figure 3.12: Flow of the creation of a `Proxy` throughout the CommonAPI components

information presented beforehand, a couple of diagrams were created that show the control flow of an application using CommonAPI. As an example, the Camera service application of the system example is used. This service needs to subscribe to Infotainment Hub `shutdown` broadcast, and thus it must create a proxy to its service. This same proxy will be used later on every time the Camera needs to send a new frame to be rendered by the Infotainment Hub by calling its method `new_frame()`. These 2 aspects are the ones that will be analyzed in further detail. Starting with the creation of the proxy to the Infotainment Hub application, an overview of the control flow is shown in Figure 3.12.

Naturally, the creation of a proxy to the Infotainment Hub service begins when calling `Runtime` `build_proxy()` method. This method receives a template argument, `T`, that represents the type of `Proxy` to be built. Additionally, it takes some identifiers as input to allow the `Runtime` to decide the exact service instance to connect the proxy to. Immediately after this method is executed, it will initialize all of the `Factories` that have registered themselves at the `Runtime`. The initialization consists of executing all the `Proxy` and `Stub` setup functions that each registered at the `Factory` singleton instance. This function registration happened when the application was first executed, before even the application entry point was executed. Once all of the `Factories` have been initialized, the `Runtime` will select the one that matches the identifiers passed to the `build_proxy()` function and execute the `create_proxy()` method. This step decides which CommonAPI middleware binding to use, as each binding will have their own `Factory` implementation registered at the `Runtime`. The `create_proxy()` function will then either create a

new `Connection` instance or fetch an existing one that will be then be passed to the constructor of `InfotainmentHubSomeIPProxy`. The constructor was generated by the code generator and will instantiate all of the services broadcasts and attributes, stored as an `Event` and one of the `Attributes` respectively. In this instantiation, the identifiers assigned to these Franca concepts in the `.fdepl` files are used. In the `Stubs` case, the method, broadcast and attributes identifiers would also be registered in their constructor. After having finished constructing its broadcasts and attributes, a newly created instance of `InfotainmentHubSomeIPProxy` will be returned to `Factory`. However, as `Factory` has no knowledge of the existence of the proxy class, an upcast will be made to abstract the object over its `someip-runtime::Proxy` behavior. The `Factory` will then execute the `init()` method, whose implementation resides in the `someip-runtime` library. Inside it, the `Proxy` will register itself at the `Connection` and the `Connection` will use the SOME/IP library to request the service instance the `Proxy` should be connected to. As soon as this is finished, the `init()` method returns and the `create_proxy()` as well. All the same, another upcast needs to be made, as the `Runtime` has no knowledge of which CommonAPI middleware bindings will be present at runtime and, as such, the object is upcast into the `common-runtime::Proxy`. To finalize the whole process, the instance now needs to be downcasted into the type the user has requested. To keep the user's application middleware independent, the type that the user should use is `InfotainmentHubProxy` which contains all the necessary methods for the user to access all the remote service resources as specified in the `.fidl` files.

With the proxy to the Infotainment Hub service created, the user's application can now execute remote procedure calls and access the service attributes. A remote method that the Camera service will be executing several times is `new_frame()` in the Infotainment Hub. The procedure followed to do that is shown in Figure 3.13, which also includes the path until the remote application offering the Infotainment Hub receives the call. The dotted arrow between the 2 `Connections` represents the transportation of the message by SOME/IP. It is irrelevant whether the 2 applications are executing in the same device or different devices, the same path will be taken regardless. Once again, the path starts at the user application level with the call to `InfotainmentHubProxy`

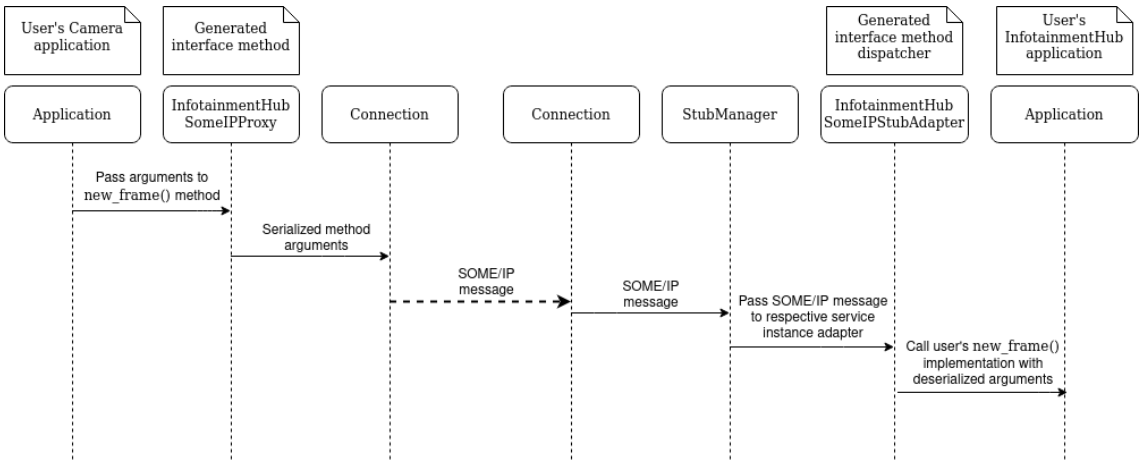


Figure 3.13: Flow of the execution of the remote `new_frame()` method

`new_frame()` method. Note that, the user's proxy object type is the generated Infotainment Hub common Proxy interface and not its someip counterpart as explained previously. However, the serialization of the method arguments is made by `InfotainmentHubSomeIPProxy` object (Figure 3.13). Having serialized the method arguments into an array of bytes, they are passed to the proxy associated `Connection` which will then pass them onto the SOME/IP implementation, via its `Application` instance, to be sent to the network. The Infotainment Hub application will then receive this SOME/IP message. Once it reaches `Connection`, if the message is a request, it is immediately passed to `StubManager` for further processing. There, the message service instance identifier will be read and passed to the `InfotainmentHubSomeIPStubAdapter`, which is responsible for that service instance. That object will then read the method/event identifier of the message and choose its corresponding method dispatcher. The method dispatcher is then responsible for deserializing the array of bytes received in the SOME/IP message payload into the `new_frame()` method arguments. This dispatcher selection and deserialization procedures are all procedures inherited from the someip `StubAdapter`. The `InfotainmentHubSomeIPStubAdapter` only needs to specify the order of the methods arguments through templated arguments, as will be shown in the next chapter. The response of the method call would then follow the same path backwards. Even when there is no return parameters, as is the case with `new_frame()`, by default, an empty response will always be sent to ensure the request was received successfully.

3.4 GENIVI SOME/IP implementation

The last piece of software required to complete the CommonAPI middleware toolchain is the actual middleware implementation. The GENIVI SOME/IP implementation, done in C++, is called `vsomeip` and was the original implementation of the protocol and, as such, follows the AUTOSAR specification [7] closely. Regarding remote communication, it is entirely compliant with the specification. However, the specification makes no mention of local communication and how it should be handled, so the GENIVI implementation has its own approach for this. The approach is centered around which application is configured to be the main application of the device. In `vsomeip`, for each device executing SOME/IP applications, only a single one of these applications will be able to communicate with external devices through the network, as seen in Figure 3.14. For simpler denomination, an application with this role will be called Host application. The remainder of the applications that are executing in that same device will have to proxy their requests and responses through the Host application. Likewise, any incoming message will also be dispatched by the Host application to the respective internal SOME/IP application offering the target service instance of the message. The decision of which application gets to be the Host can either the explicit, where, through a configuration file, it is specified exactly which application will be the Host, or it can be implicit, where the first `vsomeip` application to execute in the device will declare itself as the Host. The reason for this architecture is unknown to the author and will not be

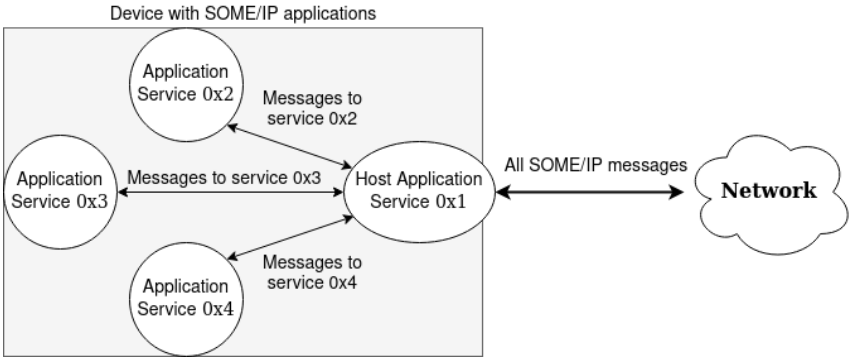


Figure 3.14: Overview of the messages exchanged between the different SOME/IP applications running in the same device

explored, as the Rust implementation needs to be interoperable with the GENIVI implementation, it will naturally follow a similar architecture.

3.4.1 Overview of `vsomeip` components

As is to be expected, the SOME/IP implementation is the most complex piece of software. Hence, an initial high-level analysis will be made here, while a more profound analysis can be found further in the document. For this initial breakdown, Figure 3.15 provides an overview of the main components of `vsomeip` and their relationships with one another.

Similar to the `core-runtime`, the `Runtime` remains the base component that serves as the entry point for a user application to create an `Application` object and then actually start the `vsomeip` procedures. That is its main purpose. It also contains methods to easily create SOME/IP messages, such as requests, responses or notifications. Although these methods are associated with a `Runtime` object, they are totally independent of it. The `Application` object seen in Figure 3.10, which `someip-runtime` `Connection` has access to, is the same `Application` that can be seen here in Figure 3.15. This is the entry point for any interaction with `vsomeip`, its public API is nearly all in `Application`. The public procedures that are available are mainly divided into 3 types, first the methods relevant for the offer and request of services as well as then sending of SOME/IP messages. Then, the offer and request of events which also includes the subscription and notification procedures. Lastly, the registration of handlers that are executed in reaction to certain `vsomeip` events, this includes the message handlers, the subscription status handlers and the service availability handlers, among others. For this latter type of methods, the `Application` is the component that stores these handlers and subsequently responsible for executing them when required. In contrast, the 2 former methods are immediately redirected to the `RoutingManager`. More than a third of the SOME/IP related code is located in the `RoutingManager` components. It is the central component that ultimately processes all the SOME/IP messages received.

There are 2 possible managers, the `RoutingManagerImpl`, which is the one that is used by the Host application with access to the external network, and the `RoutingManagerProxy`, only capable of communicating directly with the Host application. They both inherit from the

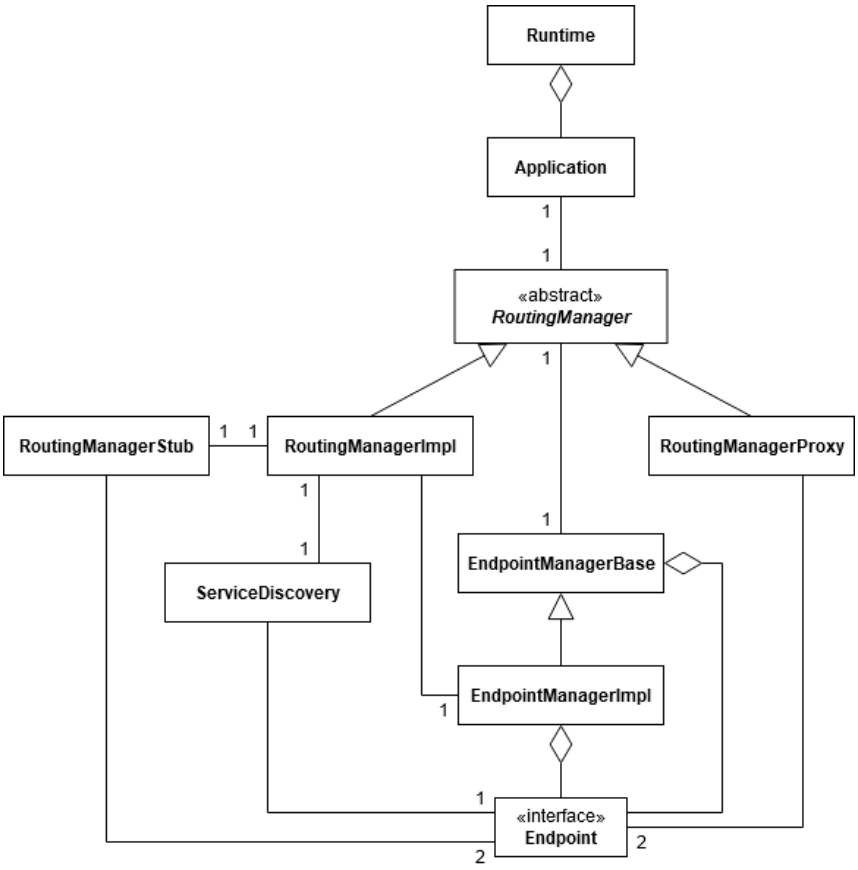


Figure 3.15: Diagram of the major classes of `vsomeip`, the GENIVI SOME/IP implementation

`RoutingManager`, which is where the reference to `Application` and `EndpointManagerBase` is stored. The application own events are also stored there, as well as its own offered services and remote services information. Methods for accessing and updating this information are also implemented in `RoutingManager`, thus being shared by both the Proxy and Host application.

The `RoutingManagerImpl` is the manager that is used by the Host application and hence needs more logic to handle both local and external communication. As previously mentioned, all of the SOME/IP messages received by the application eventually end up being processed by this component. It will then decide where to send the message, either upstream to the `Application`, downstream to `Service Discovery` or directly to an `Endpoint` i.e., `vsomeip` closest component to the network level. For the local communication, the first point of entry is the `RoutingManagerStub`, while external communication ends up directly at the `RoutingManagerImpl`.

Interestingly, in this architecture, the component will end up with 2 redundant references to `EndpointManagerImpl`, one through its data inheritance of `RoutingManager` and one present in the `RoutingManagerImpl` itself. The manager itself does not possess all of the data relevant to the routing decisions that need to be made, but it can access the necessary data through the other components. All of the data that pertains to local communication is present in both the `RoutingManagerStub` and the `EndpointManagerBase`. It does not follow the SOME/IP protocol but a protocol created by GENIVI for this specific use case that serves as a wrapper for the

actual SOME/IP protocol and also for the exchange of some additional information between applications in the same device. `RoutingManagerStub` only holds the so-called local server endpoints, serving as the receivers of local communication while `EndpointManagerBase` holds the local client endpoints for sending local messages. The latter only creates, deletes and stores these local client endpoints on demand, which is the only functionality of this component. The former initially creates 2 endpoints. One endpoint acts as the receiver for these communication protocol messages that do not contain any wrapped SOME/IP message. Instead, these are used for mimicking SOME/IP concepts, such as the registration of applications, service and event offering and many others. Some of these types of messages do not pass through the `RoutingManagerImpl` and are instead directly processed by the `RoutingManagerStub`. The other endpoint only receives the messages that actually wrap a SOME/IP message. These are usually method calls, responses or event notifications that the `RoutingManagerImpl` can then redirect to external SOME/IP applications.

For communication with external devices, the `EndpointManagerImpl` stores under which IP addresses and ports can the remote services be found and also creates, deletes and stores both the client and server endpoints to these remote services. The discovery of these external devices is made through the `ServiceDiscovery` component, responsible for processing the SOME/IP Service Discovery related communication. It has a dedicated `Endpoint`, used for both unicast and multicast Service Discovery messages, only. Once a Host application has been started, it will begin announcing its presence in the network by sending multicast messages that are received by all of the devices in the network. Through these, an external application becomes aware of this newly created application appearance and can start interacting with it.

Finally, all of the network functionality is provided by the `Endpoints`, whose implementation is behind several interfaces and abstract classes. There are the endpoints used for local communication and for remote communication, both can use either UDP or TCP, while the local endpoints additionally support Unix Domain Sockets (UDS) when the device uses a Unix system, for even faster communication between processes.

3.4.2 SOME/IP Service Discovery protocol

There is a whole separate AUTOSAR specification just for the SOME/IP Service Discovery protocol. It describes the on-wire representation of a Service Discovery message and the behavior of the protocol itself. In terms of the number of messages and their content, the protocol is not very complex. However, due to the way `vsomeip` is architecturally structured, with only the Host application receiving SOME/IP messages, including the Service Discovery ones, there needs to be plenty of additional logic to cope with this. The exchange of messages between applications and their respective reactions are shown in Figure 3.16. The Figure contains a sequence diagram between a SOME/IP service provider and a service requester with different kinds of communication protocols shown. Just the service requester reaction in different situations is shown, in practice there would be more messages also being sent by the requester into the network. All of the italic messages are sent to the SOME/IP Service Discovery multicast address while the others are sent

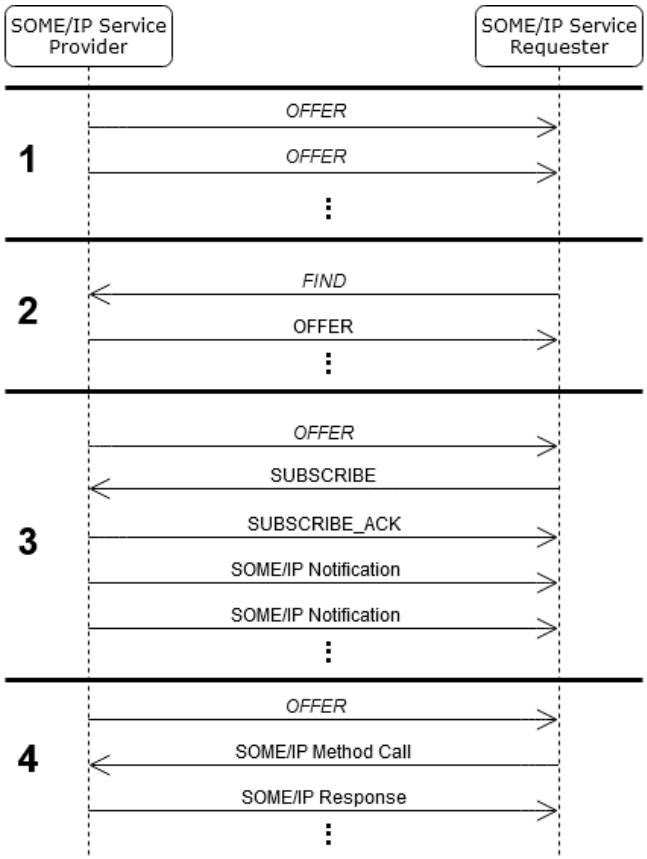


Figure 3.16: Sequence diagram of the main communication sequences of the SOME/IP Service Discovery protocol

via unicast directly to the application. While the multicast messages are always through UDP, the remainder can be either TCP or UDP according to the user specification. It is important to note that the OFFER messages contain additional information that pinpoints precisely in which IP address and port the service can be reached.

The first group of messages represents the startup behavior for the services it is offering. If the service requester is not currently requesting any of the message offered services, then it will not respond to these OFFER messages, and the provider will endlessly keep sending them. In the second group, the service requester is the one that initializes the communication procedure with a FIND message broadcasted through the network. An application that is offering the requested service will respond with an OFFER message. This OFFER message can either be sent via multicast or via unicast depending on the situation. According to `PRS_00422` and `PRS_00423` of [2], the basic implementation is just sending via unicast, while an optimized implementation may respond via either of them depending on how long the last OFFER messages from the Main phase were sent. Both Rust and C++ implementations of SOME/IP follow the optimized version.

In the third group, the SOME/IP subscribe-notify communication mechanism is shown. In SOME/IP, a client does not directly subscribe to an event but instead to what is called event-groups. Each eventgroup can contain multiple events or only a single event. Nonetheless, the

vsomeip API allows a user to subscribe to a single event of an eventgroup, however, this is merely masked by the library. What truly happens is that the application will be on the lookout for OFFER messages that contain the service instance with the eventgroup of the desired event. Once it receives one, it sends a SUBSCRIBE message to the address specified in the OFFER message. In the SUBSCRIBE message, the information about the eventgroup, only, is passed to the remote application. As such, the remote application will answer with a SUBSCRIBE_ACK message and start sending notifications for all of the events that belong to the eventgroup. The application that subscribed is then responsible for filtering these notifications and alerting the user about the events it has subscribed to, only, which is what the RoutingManager components of vsomeip do.

Additionally, the SOME/IP specification distinguishes between two types of events, the Events and the Fields, both following the subscription mechanism explained. Conceptually, an Event is meant to be used for periodic or on change notifications, while the Field is meant to be used alongside method calls representing the getter and setter of the Field. If another client would trigger a change of value in the Field, then a notification would be sent to all subscribed clients, much like an Event. In practice, both of them are implemented very similarly, except for the possibility of requesting an "Initial Event" when using a Field, where the service sends a notification with the Field value immediately after the SUBSCRIBE_ACK message.

The last group shows SOME/IP request-response communication mechanism for remote service found using Service Discovery. After the OFFER message is received and the service is offered in that message, the client starts sending requests to which the provider responds. If a static IP address and port of the remote service is specified in the vsomeip JSON configuration files, it is possible to initialize the request procedure without having to wait for an OFFER message. When using TCP, as it is a reliable end-to-end protocol, it is ensured that the remote application will receive the messages as it first needs to wait for it to be available and listening on the remote address.

3.4.3 Configuration of the SOME/IP protocol

Always accompanying a vsomeip application are the configuration files. These hold information crucial to the execution of a vsomeip application. Without one, it is only possible to have local communication. These files follow the ubiquitous JavaScript Object Notation (JSON) and can be composed of several elements. It is possible to specify either a single file or a folder containing several files and vsomeip will then merge the configurations. They will all get parsed into an additional component not shown in Figure 3.15 named Configuration. All of the components shown in that Figure have access to a shared Configuration instance, allowing fast access to its parameters. Due to the sheer amount of possible configuration possibilities, only those required for communication with external devices will be analyzed here and shown in Figure 3.17.

The most basic parameter is unicast, containing the IP address of the network interface the application will be bound to. This is mainly relevant for systems where multiple network interfaces may be available to ensure that they bind to the desired one. Below that, there is the applications JSON array element that holds an arbitrary number of JSON maps corresponding

```
{
  "unicast": "192.168.1.1",
  "applications": [
    {
      "name": "Infotainment Hub",
      "id": "0x1000"
    }
  ],
  "services": [
    {
      "service": "0x1234",
      "instance": "0x5678",
      "unreliable": "30509"
    }
  ],
  "routing": "Infotainment Hub",
  "service-discovery": {
    "enable": "false"
  }
}
```

Figure 3.17: Extract of a `vsomeip` configuration file with the main configuration parameters

to the specification of a single SOME/IP application. For each of these maps, the mandatory keys are `name` and `id`, the former holding the name of the `vsomeip` application mainly relevant for logging purposes while the latter is crucial for the communication with external devices. Each SOME/IP application in a system needs to have a unique identifier, specified by this `id`. If this is not specified, then each application will be automatically assigned an identifier and report it to the device Host application. This is perfectly fine for local communication, as this assignment and registration with the Host application ensures that each application will be assigned a different identifier. However, remote applications have no way of knowing the identifiers of all applications in the network. As such, unless specified in a configuration file, 2 applications may be assigned the same identifier and thus causing the protocol to misbehave.

Next, there is the `services` key, that follows a similar approach to `applications`, containing an array whose elements are maps corresponding to a single service instance. Each of them needs to have the `service` and the `instance` keys in order to distinguish to which service instance the element is referring to. Then, there are several additional possible keys present, but the most crucial ones being either the `unreliable`, seen in Figure 3.17, or the `reliable`. The information both of these hold corresponds to the network port through which the service can be accessed externally, and without it, the service will not be reported to external devices. A not so crucial but relevant to mention parameter is the `routing`. This key holds the name of the `vsomeip` application, as specified in the `applications` array, that will become the Host application and thus be able to interact with the network. While this key is not necessary for either local or external communication, since without specifying it the first application that executes will become the Host application, it is often specified.

Finally, there is the configuration section of SOME/IP Service Discovery protocol `service-discovery`. In this example, the Service Discovery is turned off. However, several possible parameters can be used here to customize the behavior of Service Discovery that will be analyzed further in the document as part of the Service Discovery description. Do keep in mind that without Service Discovery enabled, it is impossible to have a subscribe-notify type of communication between remote services. Nonetheless, it is possible to have request-response, but it requires the specification of additional parameters in the configuration file to pinpoint exactly to which remote IP to send the request, but this defeats the purpose of having a SOA.

3.5 Summary

This chapter introduced a high-level analysis of the CommonAPI toolchain and how all of these tools interact together. The chapter included the description of an example system, that portraits a simple use case for a SOME/IP powered vehicle with devices subscribing to each other events and then reacting accordingly, either with an update of the local information or with the triggering of remote procedures from a third device. Then, we walked through the steps required to transform the abstract system description into an actual implementation of the system. This started initially with the definition of the system through the system definition language Franca IDL. Afterwards, a breakdown of exactly how each of the example system concepts are mapped into Franca was shown and also some additional capabilities not used in this system. Having fully defined our system in Franca, we then moved into how the CommonAPI code generator parses these files and maps them into their respective code implementation already with the separation of the service clients and the service providers. This mapping is supported by the CommonAPI Runtime, which provides an abstraction of the Franca concepts over the underlying middleware to be used. As such, we made an initial analysis of the CommonAPI components, including both the core Runtime that merely serves as the abstraction layer, and the CommonAPI SOME/IP Runtime that provides the underlying implementation of the CommonAPI and Franca concepts and their respective translation to SOME/IP concepts. Beneath all of these abstractions, the chapter ended with a high-level look at the GENIVI SOME/IP implementation, `vsomeip`, with a dissection of its components, how they interact with each other and also an explanation of how to properly configure a `vsomeip` application.

Chapter 4

Rust implementation

Having already taken a higher-level view of the CommonAPI toolchain, its workflow and inner workings of each tool, this chapter presents an implementation perspective of each of these, using the Rust language, as well as a comparison between the original reference implementations and the implementations done in this work. Similarly to the previous chapter, the analysis will focus first on the code generator, then the CommonAPI Runtime and finally, the SOME/IP implementation.

4.1 CommonAPI Rust code generator

Very early in the planning of this work, the question arose of whether to do a fully separated Rust code generator, or to adapt the existing ones to produce Rust code instead of C++. We opted for the latter approach as it seemed the one that would be more time-effective, as by building on top of an already existing project, there were already certain aspects that could be taken for granted. A good example of this is the parsing of the Franca files, the existing generators already handled this, so by building on top of them, that is no longer a feature that needs to be implemented. The basis for the Rust generator was version `3.1.12.4` of `core-tools` and `3.1.12.2` of `someip-tools`. In the meantime, version `3.2.0` of both tools was released, only partial support for this release was added to the Rust generator, namely the support for version `0.13.1` of Franca and the upgrade to the newest version of the CommonAPI runtime. The generators were built using a mixture of Java and Xtend, a language developed by the Eclipse Foundation, based on Java, which provides additional concepts that make it simpler to develop code generators and ultimately ends up compiled into Java, thus integrating seamlessly with other Java code or libraries.

An immediate difference that can be noticed between the Rust code generator and the C++ ones is that there is only a single executable, as shown in Figure 4.1. While the C++ implementation initially uses the `core-tools` generator to generate middleware independent code, which is then complemented by the middleware specific generators, the Rust generator is a merge of both `core-tools` and `someip-tools` which uniquely generates code that uses the CommonAPI SOME/IP Runtime. The reason for this is that the original planning of the work did not include implementing the CommonAPI capability of supporting multiple middlewares and so a

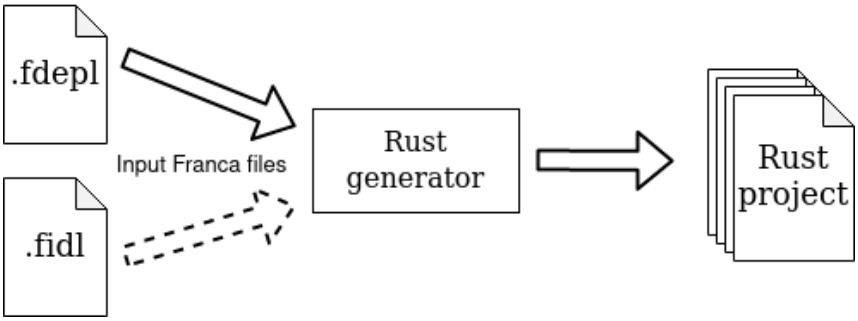


Figure 4.1: Overview of the rust code generator inputs and outputs

single generator would have been required. Due to this, the Rust generator input is the same as `someip-tools`, it only accepts `.fdepl` files. Despite this, the generator can still access the respective `.fidl` file, since the `.fdepl` files have a reference to it, hence the dotted arrow in the Figure. Both of these files initially go through both the `core-tools` file validator, for the `.fidl` files, and the `someip-tools` validator for the `.fdepl` files, guaranteeing that the error reporting matches the reference implementation. If this verification finds no errors, it starts the actual code generation by traversing the `.fidl` file and translating the abstract concepts into concrete code. Using the Franca files from the example system presented in Chapter 3, passing the option to generate the skeleton, thus executing the command

```
> ./commonapi-generator-linux-x86_64 -sk CompleteCar.fdepl
```

a folder named `rust-gen` will be created in the directory where the generator was executed. That folder will contain the code generated for the passed `.fdepl` file, present in Appendix B.2. Inside it, the generated code and folder structure will be present and can be seen in Figure 4.2. The structure follows the traditional Rust workspace, where at the root of the workspace there is a `Cargo.toml` file that contains different kinds of metadata of the project, like the author of the project, its version and its dependencies. All of the code is located in a `src` folder, where usually a `main.rs` file marks the entry point for the binary. However, it is also possible to define several binaries for a single workspace as long as their names and entry points are specified in the `Cargo.toml` file. As the generator has no information on the whereabouts of the Rust SOME/IP implementation and the Rust CommonAPI Runtime, it uses default values for their dependency declaration that then need to be updated by the user. Some additional dependencies are present that should not be removed and, of course, if the user requires any additional dependency for its project, it needs to be added in the `Cargo.toml` file. The full generated `Cargo.toml` file can be found in Appendix B.3.

Since the generated code assumes nothing about what the user is trying to achieve, it just presents code as a library that the user's application can use. The generated code is stored under the `lib` folder and is declared as a library in a `[lib]` section in the `Cargo.toml` file, thus making it available to all of the binaries in the workspace. At the root of each folder that contains Rust code, a `mod.rs` file needs to be present, as shown in Figure 4.2. Rust uses these to declare

```
rust-gen
|   Cargo.toml
+-- src
    +-- lib
       | mod.rs
       +-- CompleteCar
           | mod.rs
           +-- common
           |   +-- v1/elektrobit/rust
           |       | mod.rs
           |       | AutomotiveTypes.rs
           |       | Camera.rs
           |       | Camera_someip_deployment.rs
           |       | ...
           +-- client
           |   +-- v1/elektrobit/rust
           |       | mod.rs
           |       | Camera_proxy.rs
           |       | Camera_someip_proxy.rs
           |       | ...
           +-- service
               +-- v1/elektrobit/rust
                   | mod.rs
                   | Camera_stub.rs
                   | Camera_someip_stub_adapter.rs
                   | ...
```

Figure 4.2: Resulting generated files and directory structure using the Rust generator on the example system showing only the files for the `Camera` service and the type collection `AutomotiveTypes`

a code module and child modules to provide code modularity. Then, for each interface present in the `.fidl` file, a folder is created where the code for that respective interface can be found. In the `CompleteCar.fidl`, only a single interface `CompleteCar` was declared, and thus only that folder is present. Inside it, again the `mod.rs` file, then the `service` folder, containing code related to the service provider part of the interface, a `client` folder for the service client part of the interface and then a `common` folder with code shared by both the service and client code. Each of these has several inner directories where the first is the version number of the interface, and the subsequent directories are the declared package of the interface. As the example system uses `package elektrobit.rust` and version 1.0 it produces the `v1/elektrobit/rust` folder structure. Each of these inner folders has a `mod.rs` folder at its root, not shown in Figure 4.2 for simplicity purposes. At the end of all of the inner folders, the actual Rust generated files are present. These follow the same structure as the C++ code generators, with a non-middleware dependent file, and an equivalent SOME/IP specific file, all with the same responsibility as their C++ counterparts. An additional file, `AutomotiveTypes.rs`, in the common code folder is also shown here to show how a Franca `typeCollection` declaration translates into concrete code. Seeing that a type collection contains types that can be accessed by all of the interfaces, it naturally goes into the common code folder. This follows the same principle as C++ where a file

is generated for each type collection.

Similarly to the C++ code generator, the Rust code generator allows the user to specify a number of optional flags to the executable. For example, in Figure 4.2, the "skeleton" flag (`-sk, -skel`) was activated. This is likely the most relevant flag as it causes the generation of additional helper code. Another interesting flag is the "search path" flag (`-sp, -searchpath`) that allows the user to specify a directory that contains Franca files. The generator will then parse all of the Franca files of the directory and generate the respective code, thus allowing multiple `.fidl` and `.fdepl` files to be used as input. The remaining of the flags are either related to changing the output directory of the generated code or disabling sections of the generation altogether, such as disabling the generation of the common code.

4.1.1 Differences to the reference implementation

Although the Rust code generator tries to follow the reference implementation as closely as possible, since the target language is different there are some differences between the generators. The most basic one is in the behavior of the generator when using the "skeleton" flag. When it is turned on in the `core-tools` code generator, it generates additional `StubDefault` classes, which are a default implementation of the services Stubs. The user's own implementation will then inherit from this default implementation, which allows the user to only override the specific methods it wants to implement. The remainder will use the default implementation provided by the generator. This default implementation ensures that the getters and setters of the attributes simply fetch and update the attributes, only, without any additional logic, whereas the method calls immediately return, passing default values as output parameters, for example, 0 when an output parameter is a number.

In the Rust code generator, since Rust approach is composition over inheritance it does not have the same inheriting possibilities offered by C++, meaning that even if the Rust generator would generate default implementations for the service methods, the user would still have to implement the Stub methods manually. One possible approach here would be to still provide a default implementation, and if the user wants to use it, it would have to store an instance of the default classes and then redirect the method calls to this instance. While this approach is feasible, and a similar mechanism is used in the Rust implementation of the CommonAPI Runtime, that was not the approach taken here. The reason for this is that even if the user would actually store an instance of this default implementation in its own stub implementation, it would still have to declare and provide an implementation, albeit just a redirection, for all of the methods in its own implementation. Thus the approach taken was to simply not provide any default implementation.

The Rust code generator still generates these default stub implementations, but the method calls are filled with the Rust `todo!()` macro, that causes the program to exit when called. They are simply there to provide the user with a correct example of exactly which interfaces and methods the user needs to implement in order to provide a stub implementation to the CommonAPI Runtime.

C++ generator	Rust generator
FooStub • *fireXAttributeChanged()** • *lockXAttribute()** • *getXAttribute()* FooStubAdapter • *fireXAttributeChanged()** • *lockXAttribute()** FooStubRemoteEvent • *onRemoteSetXAttribute()* • *onRemoteXAttributeChanged()*	IFooStub • *fire_X_changed()** • *get_X_lock()* IFooStubRemoteEvent • *remote_get_X()* • *remote_set_X()* • *on_remote_X_changed()*

Table 4.1: Comparison of generated function calls between the C++ and the Rust code generator for the `Foo` service with an attribute `X` with no access restrictions. Marked with an * are the methods already implemented by the generator.

Additionally, in the Rust generator, the "skeleton" flag also generates the default `Cargo.toml` file, a crucial part for any Rust project. It does this in order to provide the user with an example of how should the Cargo file be built. The user is later free to modify it. Accordingly, it is only expected that the user executes the generator with the "skeleton" flag once. Otherwise the Rust generator will overwrite the `Cargo.toml` file and, if the user did any change to it, they would be lost.

Of course, these methods that the user needs to implement are generated for either a specific method present in the service description, an attribute or a broadcast. While the method and broadcast generated methods are the same between the C++ generator and the Rust one, this is not the case for the attribute related methods. A comparison of these generated methods between both generators, based on a service named `Foo` with an attribute `X` without any access restrictions, can be seen in Table 4.1. In it, not only are the generated methods shown but also the language concept to which they are associated, in the C++ case a class and in Rust an interface, and also which of the methods are already implemented by the code generator, marked with an *.

As can be seen, the C++ generator generates an additional class, `FooStubAdapter`, that contains methods that the middleware specific stub adapters have to implement. This is not for the end-user to worry about as it will only be used by the remaining generators. Hence, the two classes that the user needs to consider are the `RemoteEvent` and the `Stub` classes. The user will have to implement the methods in those two classes if correct Stub implementation of its own is to be provided. Since the Rust generator merges both the core generator and the SOME/IP generator,

it does not need this additional `StubAdapter` interface as it is all generated in one execution. Other than that, the Rust interfaces match their C++ equivalent, but the methods inner logic may vary. The Rust methods naming convention is different from the C++ ones to match Rust official naming conventions.

The most glaring difference is that in the C++ generated code, the attribute `fire` and `lock` exist in both the StubAdapter and the Stub. For the `fire` method, the one in the former class is the one that will actually call the respective CommonAPI Runtime procedure to then send the event notification to the subscribers, while the one in the latter class is simply a redirect to the StubAdapter equivalent method. Whereas in the Rust generator, this indirection is removed since the Stub has an instance of the StubAdapter and through it access to the CommonAPI Runtime procedure, so it does the call directly. From a compilation point of view, under optimized builds, this will probably make no difference as it is likely that the compiler will inline the C++ indirection, thus removing any runtime cost. However, the indirection can become confusing for a user that seeks to understand how the generated code works.

The other method, `lock` is quite interesting and fairly different between the C++ and the Rust version. It exists to synchronize accesses to the attribute and, in C++, it receives a boolean as input that signals whether to lock access or unlock it. Similarly to the `fire` method, the Stub `lock` method is simply a redirect to the one in StubAdapter, where the actual mutex primitive is stored under the member `std::recursive_mutex xMutex_` for the attribute `X`. Before the StubAdapter calls either the getter or the setter methods of the Stub for the attribute, it locks the mutex, thus ensuring that even if the same request is received, only one will be processed at the same time and no data races happen.

This approach is perfectly fine for the code that has been generated as it guarantees memory safety. However, once we consider the user application space, it becomes problematic. The user's own implementation of the Stub is where the actual member with the attribute value will be stored, so we end up with the attributes mutex in the StubAdapter and the actual attribute in the user's own Stub implementation. Already this aspect is strange, as commonly the mutex for a variable resides in the same place as the variable itself so the user can easily identify where exactly the synchronization mechanism is taking place. Nonetheless, the more crucial aspect is that because the attribute value resides within the user's own class, the user has no restriction over when or how it can access this value. As such, the user could be writing to the value while another thread spawned, in response to a request, would also be writing to the same value, effectively causing a data race. In order to mitigate this, the C++ code generator default Stub implementation, generated when the "skeleton" flag is passed, provides an additional setter method for each attribute. The proper synchronization mechanisms are used inside that method, initially locking the mutex with the `lock` method, then actually changing the attribute value, executing the `fire` function and finally unlocking the mutex again. As long as the user remembers this detail and takes the time to understand how the synchronization takes place, there should not be a problem. However, it is undeniably an approach that may confuse some users and cause issues for a more distracted user that did not take the time to properly read the documentation and how to develop applications

```
fn main() {
  let foo: Arc<usize> = Arc::new(10);
  let bar: Arc<Mutex<usize>> = Arc::new(Mutex::new(10));


  *foo = 0; // Operation does not compile, cannot mutate Arc directly
  *bar.lock().unwrap() = 0; // Allowed since inside the Arc we use a Mutex
}
```

Figure 4.3: Snippet of Rust code demonstrating inability to directly mutate a shared pointer.

using the CommonAPI toolchain.

Due to Rust memory constraints and how it handles synchronization primitives, this issue is entirely eliminated, and the user cannot unintendedly cause any data races. Since the user's Stub implementation will be shared between different threads, both the C++ and the Rust implementation need to encapsulate it in a shared pointer. In C++ this translates into a `std::shared_ptr`, while in Rust its called an `Arc`. The main difference between the two is that the Rust version restricts mutable access to the shared pointer data. Under safe Rust, a user cannot mutate the data in an `Arc` directly. This is demonstrated through a simple example in Figure 4.3,[1] where the compilation of the code fails due to the inability to mutate an `Arc` directly.

Taking this into account, in the Rust implementation, we naturally cannot directly mutate the attributes of this Stub implementation. In order to provide mutable access to the data inside the `Arc`, additional synchronization mechanisms need to be used, either mutual exclusion primitives such as a mutex, or, in the case of integers, through atomic variables. Since we want the exclusion mechanism, even for an integer, mutexes are used for all attributes. Unlike C++ mutexes, Rust mutexes need to have an associated object which will be the only object that can be mutated while holding the mutex lock. For example, the declaration of a mutex holding a boolean would be `Mutex<bool>`. Naturally, a mutex can hold much more complex types such as structures or maps. Considering this and applying it to the code generated by the Rust generator, we can see how it becomes impossible for a user to cause a data race, even unwillingly, as the user is forced to lock the mutex in case it wants to change an attributes value. However, unlike C++, there is no data inheritance in Rust, only method inheritance, so the generator cannot simply create this mutexed attribute and is dependent on the user's Stub implementation to do it, so a different approach had to be taken.

Unlike C++ `std::recursive_mutex` that returns nothing when it is locked, when Rust `Mutex<T>` is locked it returns a `MutexGuard<T>` that provides access to the associated object and will unlock the mutex once the guard falls out of scope and is thus destroyed. Taking advantage of this functionality, we can force the user to actually store the attribute value inside a mutex by requiring the Stub method `get_lock` method to return a `MutexGuard`. Since there is no other way to acquire it other than to lock the mutex, the user needs to encapsulate the attribute value

[1] A compilation attempt can be made here

in one. This method is then, similar to the C++ implementation, used throughout the getters and setters for that attribute to ensure mutual exclusion. The user is free to either call the `get_lock` method or directly lock the mutex since it has direct access to it.

The other methods and their functionality are similar between C++ and Rust. As for the setter logic, in C++, they initially call the respective `set` method, which will return a boolean representing whether the attribute was actually changed or not, the decision is left up to the user's implementation. A response is immediately sent back to the client that initiated the setter procedure, containing the attribute value, whether updated or not and then the attribute mutex is unlocked. In case the attribute was changed, the lock for the attribute is reacquired, and a notification to all of the remaining clients subscribed to the attribute is sent. Afterwards, the mutex is unlocked, and the Stub `changed` method is called so that the Stub can do any local work after the attribute value has been changed. Rust approach changes the order of events, instead of initially sending the attribute value back to the client that triggered the setter procedure, it first sends a notification to all of the subscribed clients, calls the `changed` method and then finally sends the response back to the requesting client. The reason for this change is related to the split serialization logic between the Rust code generated and the Rust CommonAPI Runtime and will also be explained further on. Nonetheless, no practical difference should arise due to this change in order.

In the getter procedure, the logic changes a bit. As seen in Figure 4.4, in the reference implementation, the attribute mutex is initially locked, then the Stub `get` function is called which returns a reference to the attributes value, then the attribute mutex is unlocked, and the attribute value obtained through that reference is serialized and sent back to the requesting client. An issue can already be found with this approach, even assuming that the user always follows the mutex locking/unlocking procedure to modify the attribute value. The fact that the mutex is unlocked before the reference returned from the `get` method is used, means that there is a race condition here. It is possible for the user to change the value or even destroy it, in the case of an allocated data type as a vector or map. The first case is not so problematic, as the value that is serialized merely is different from the value initially returned by the `get` method. However, the second case is much more severe and can either cause a segmentation fault or be undefined behavior. Of course, this is a type of error that will only happen once every millionth execution or even less. Regardless it could easily be avoided. These memory issues are precisely where Rust shines with its very strict compiler checks that ensure memory safety. In the Rust version, the attribute value serialization needs to be done while the lock is held, the Rust compiler would not allow a reference to the value otherwise.

An additional feature was added as an optional response. Whereas in C++ the `get` method necessarily returns the attributes value, in Rust this was made optional. In case no value is returned then the server replies with an error message which the client can recognize. To aid the server in this decision, the message client identifier is passed to the getter, much like how it is done in the setter.

```
procedure get_foo(message, stub):
  client = message.client
  stub.lock_foo(true)
  value = stub.get_foo(client)  // Reference to actual value
  stub.lock_foo(false)
  reply = value.serialize()
  send_reply(reply)
```

Figure 4.4: Pseudocode with the logic for fetching the value of an attribute as seen in C++ `StubAdapters`

4.1.2 Rust code generator limitations

Not only does the Rust code generator have differences from the reference generator, but also has certain limitations, some inherited by the reference generator and some exclusive to the Rust generator. Starting by the inherited limitations, since the reference generator version that was used as the base for developing the Rust generator was `3.1.12`, of both the core generator and the SOME/IP generator, and in this version support for Franca constants was still not fully added it also did not make it into the Rust generator. Although their declaration is being parsed and validated, there is no resulting code whatsoever to match it. In the more recent version of the reference generators, `3.2.0`, this appears to have been fixed. As previously mentioned, the Rust generator only has partial support for version `3.2.0` of the reference generators. This is due to this version only having been released late in 2020 and also not compiling at that time, due to a missing folder, `src-gen`, the author had forgotten to upload. [2] Despite this being an easy fix, it was only added over a month later, further invalidating the possibility of that version being included in this work.

The other Franca concept not supported by the C++ code generators are Franca contracts. These establish a Protocol State Machine that each client connected to the service needs to follow. It creates several states and state transitions that restrict the client access to the service functionalities only to the ones allowed by the current state that the client is in. This could be used, for example, to force a client in an "idle" state to call the method "wakeUp" to change its state. Right as the planning of this work was being made, it was already decided that we would not be supporting these Franca contracts as they seemed to be a very specific feature that is logically completely different from what the Franca IDL tries to achieve. Only later, once the development actually began was it realized that the GENIVI code generators also do not support this Franca feature and so it posed no problem.

A single Franca concept has not been implemented in the Rust code generator that exists in the C++ one, the "Interfaces managing interfaces". As the name implies, this establishes a main interface responsible for the interfaces it is declared to manage in the Franca specification. The Franca specification does not specify exactly what this relationship means, and whether there should be any additional restrictions for the communication between external services and these managed

[2] According to issue #25 and #26

interfaces, it merely conveys a relationship between interfaces. Additionally, in the Franca specification, it is mentioned that this concept has been extracted from some IPC mechanisms, namely D-Bus, the other middleware supported by the GENIVI CommonAPI. Knowing this, and also considering that this thesis work is only focused on the SOME/IP part of the CommonAPI, it did not seem much of a relevant feature, and as such, it was dropped.

4.2 Rust CommonAPI Runtime (`capi_runtime`)

The CommonAPI Runtime serves as the translator from the Franca concepts to their respective concrete middleware representation. Any program using the CommonAPI toolchain inherently uses the Runtime. This means that, unlike the code generators, this tool has a direct impact on the non-functional requirements of a project and, as such, it should be as lightweight and easy to use as possible. That is precisely the case with the C++ CommonAPI Runtime, where indeed there is only elemental logic in the tool, but nonetheless, it is still a complex tool from an architectural point of view, as it needs to be able to support multiple abstraction levels. From its heavily templated functions and classes to the multiple inheritance levels, one must go through several indirections until the piece of code implementing a functionality can be found. Already from the beginning of the Rust implementation, it was evident that the limits of the Rust typing system would have to be explored and workarounds would have to be taken to translate the concepts possible in C++ to Rust.

For an initial high-level overview of the Rust CommonAPI Runtime components, a class diagram comparable to the one shown in the previous chapter in Figure 3.10 can be seen in Figure 4.5. Similarly to the reference implementation, the project is separated into the common code, shared by all middlewares mostly composed of interfaces, and the someip code which is the implementation of the common code for the SOME/IP protocol. There is only a single Rust CommonAPI Runtime library, named `capi_runtime`, where each code section is separated into a different module, so the common code is in the `common` module, whereas the SOME/IP code is in the `someip` module.

In the diagram, the relationships and elements in bold represent a new or different concept to what could be found in the reference implementation shown before. Some relations that are not shown for simplicity purposes are the `someip` attribute objects implementation of their respective `common` interfaces, so the someip `ReadonlyAttribute` implements `IReadonlyAttribute`, `Attribute` the `IAttribute` and so forth. This is a much simpler and straightforward approach than the one taken by its C++ counterpart where their templated inheritance can lead to some quite complex types. The remainder of the changes are related to the Rust paradigm of "composition over inheritance". Languages that opt for the inheritance approach, like C++, allow the definition of an abstract base class. In it, both shared behavior and shared data can be defined that are subsequently inherited by objects extending it. The base class can then be used throughout the code to abstract over the concrete implementation by dynamically dispatching it. This type of abstraction is also possible within Rust, albeit more limited. It is impossible to define abstract

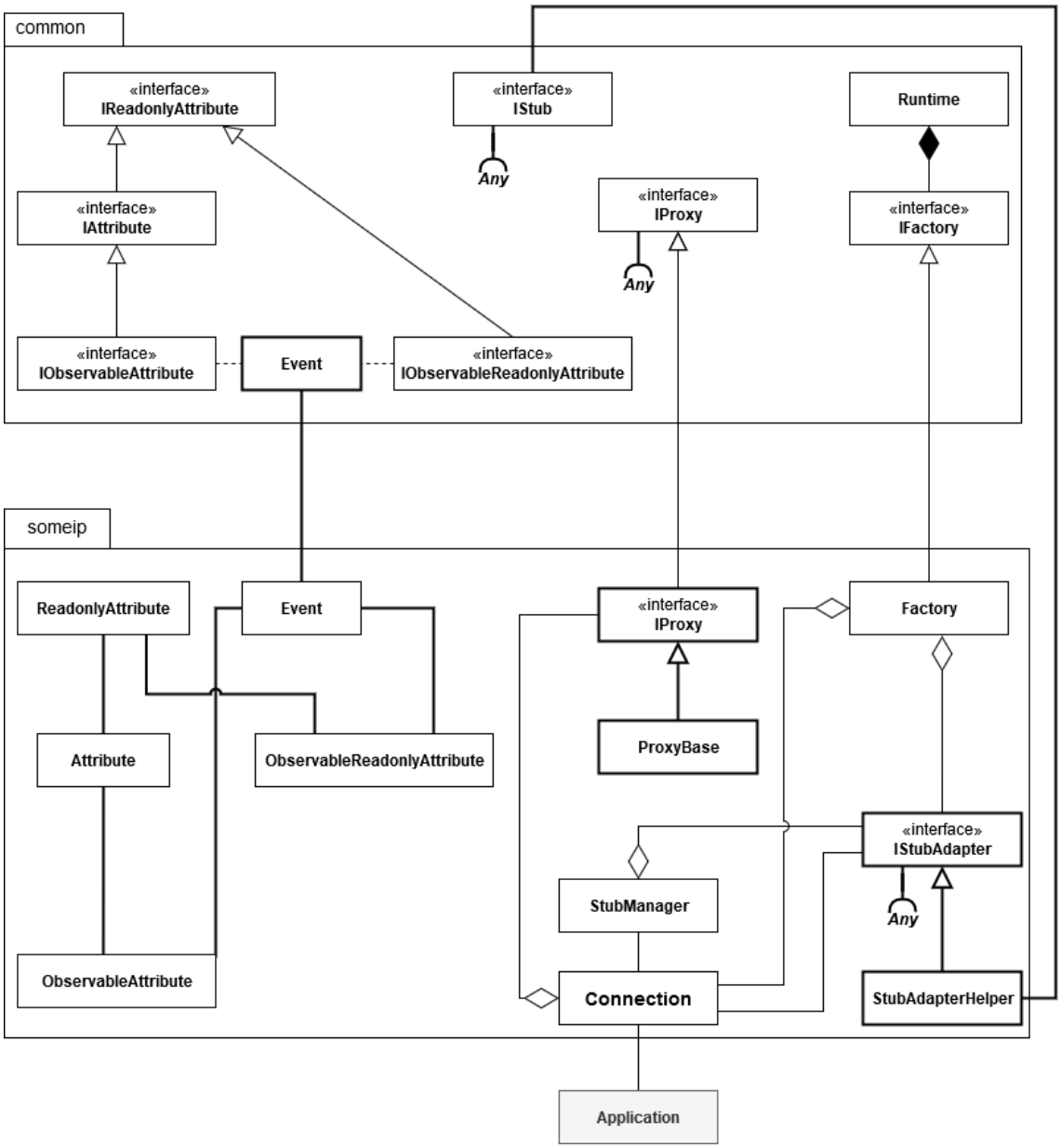


Figure 4.5: Diagram of the major classes of the Rust CommonAPI Runtime implementation, named `capi_runtime`, which also includes the SOME/IP Runtime under the `someip` module

classes, only interfaces, named `traits`, that specify shared behavior without any associated data member. The storage of any data needs to be within a concrete object that can be instantiated. If an object wants to "inherit", in the C++ sense, both data and behavior from another object it needs to store an instance of it.

This approach is exactly what can be observed in multiple relationships. For example, the common and someip modules `Events` have a regular object, instead of an `Event` abstract class in the common module. Then, the someip `Event` only needs to store an instance of this object. The same concept was used for the someip attributes, the base object with the common behavior and data shared by all attributes is the `ReadonlyAttribute`. Then, with each composition, more functionality is added on top of the already present, so the `ObservableAttribute` stores an instance of `Attribute`, which in turn stores an instance of `ReadonlyAttribute`.

Another glaring difference is how the common `StubAdapter` interface has been removed. While the interface has indeed been removed, the `IStub` interface is still required to store an instance of a `StubAdapter`. The reason for this removal is that the interface only had a single getter method and was not used anywhere else in the code with the use of `Any` replacing it.

The same type of composition is also applied to the Proxies and StubAdapters objects, but it needs an additional twist when compared to the attributes. Unlike those, it is known that a user will be providing their own implementations of `Proxies`, `StubAdapters` and `Stubs`. In practice, the end-user needs, only, to implement `Stubs`. The remainders are implemented by the code generated through the Rust code generator. Nonetheless, from this library perspective, both implementations belong to the user's application. In this aspect, the possibility of defining and extending an abstract class in C++ is a great aid, as the user simply extends the class and will inherit both its behavior and data, whereas composition still implies that the user needs to implement methods with the stored object instance behavior. That is why it is required for the C++ abstract classes `Proxy` and `StubAdapter` to be split into 2 in the Rust version: 1) the interfaces, `IProxy` and `IStubAdapter`, that specify the shared behavior, and 2) the object with the implementation of these interfaces, `ProxyBase` and `StubAdapterHelper`. Then, the user's implementations of the `Proxies` and `StubAdapters` just needs to store an instance of these objects and redirect all of `IProxy` and `IStubAdapter` calls to it.

While this design of splitting an abstract class into 2 certainly works, it poses another minor problem. The user is still required to manually write an implementation of the interfaces for its object that simply redirects to the instance of the `capi_runtime` someip objects. This is not a problem in the case of the `Proxies` and `StubAdapters` as their implementations will be created through the Rust code generator and as such, the actual end-user needs not worry about it. However, the same is not true for the implementation of `Stub`. As the end-user is the one that actually provides the `Stub` implementation, the `IStub` methods would have to be implemented manually. Although there are 2 simple methods, only, that need to be implemented, 1 getter and 1 setter, it is still non-transparent to the end-user, and it would be better to just abstract the user from it.

To mitigate this and make it simpler for the end-user a couple of procedural macros were

```
#[derive(Capi_IStub)]
pub struct WheelsService {
  adapter: Mutex<Option<Arc<dyn Any + Send + Sync>>>,  // StubAdapter
  ...
}
```

Figure 4.6: Snippet of a user implemented Stub using `derive` macros to automatically implement `IStub`

developed that automatically implement the interface for the object to which they apply to. These are called `derive` macros, and they are a powerful tool used throughout the Rust ecosystem. Its usage applied to an implementation of the `Wheels` service in the example system can be seen in Figure 4.6. It follows the same principle as Rust feature flags and only applies to the element directly after it. The compiler will then expand the macro and automatically generate the code that implements `IStub`. The member `adapter` that can be seen is where the object actual instance that implements `IStubAdapter` will be stored.

4.2.1 Middleware independence

A crucial part of the CommonAPI is the ability to abstract the end-user of the middleware that will be used by the application, thus allowing the user to swap middlewares easily. In that sense, the CommonAPI follows an approach reminiscent of a SOA, making it even more suitable for these kinds of systems. Consequently, the Rust implementation also aimed to provide this abstraction, although it was a secondary goal and priority being given to both implementations interoperability and that Rust implementation has at least the same runtime characteristics as the reference implementation. Nonetheless, full middleware independence was achieved in the Rust implementation. To do this, some modifications had to be made, as workarounds for the differences in the languages type system.

As shown in Figure 4.6, the user's `Stub` implementation stores an instance to the `StubAdapter`. However, the object that can be seen in the Figure is `dyn Any`. The `dyn` is just a Rust prefix used to identify places where dynamic dispatch will be used easily. `Any` is Rust sole construct that enables runtime reflection and thus allowing testing whether the instance is of a specific type or not. It is through it that it is also possible to downcast from an `Any` to the concrete object that initially got upcasted into an `Any`. Most of Rust types and user-defined types implement `Any` and are a candidate to be upcast. The class diagram in Figure 4.5, shows precisely which library types have the `Any` trait activated and are thus eligible for upcasting and downcasting from it. These are the types that need to be implemented in the user application space, either by the code generator, for the `IProxy` and `IStubAdapter`, or by the user itself in the case of `IStub`.

Therefore, multiple levels of abstraction are required. First, the `someip` module needs to receive someip `IProxy` objects to properly process them, where we are still bound to a middleware but abstracted from the user's implementation. Then, pass them onto the common module as common `IProxy` where the middleware information is lost as we are now abstracted from it. This

same process could be used for either of the 3 interfaces. In practice, this object passing through multiple abstraction layers is seldom needed, only in the actual construction of these objects do they go through these multiple layers. This is because they are built through an implementor of `IFactory` which specifies the methods to both create proxies and register the user's `IStub` implementations.

The creation of Proxies is the simplest procedure. Through a templated `build_proxy` method, the user can create instances of a Proxy associated with a specific middleware, but the user is abstracted from the concrete implementation. This method receives some identifiers that allow the CommonAPI to pinpoint which middleware to use for this specific Proxy instance and will find the `Factory` capable of building it. Once that is found, then a new `Connection` is created and passed to the `Proxy` constructor, at which point the proxy is now built and initialized. The newly created instance goes back into the initial `build_proxy` function, fully abstracted from the middleware, and then it attempts to downcast it to the type that the function received as a template argument.

This approach follows very closely with the reference implementation. The difference lies in where the `Proxy` instance is initialized. In C++ this happens immediately after it has been constructed, inside `Factory create_proxy()` function, whereas in Rust it still happens inside the constructor, hence why it was renamed to a setup function instead of constructor. This was made in order to circumvent Rust typing system, as the `init()` method is a someip `IProxy` method, so in order for the `Factory` to execute it, it would need a `dyn someip::IProxy` object, which would then have to additionally be upcasted to `dyn Any` in order to be passed to the initial `build_proxy` function. There is more liberty in C++ for upcasting and downcasting, so that was the approach taken in that implementation. However, in Rust if we were to do the same methodology, when we would then try and downcast `dyn Any` into its specific type we would receive a `dyn someip::IProxy` object as it was the type that was upcast to an `Any`. Therefore we would lose all information about the user's application typing. By moving the `init()` function call to the `Proxy` constructor, we can immediately return the `dyn Any` object obtained by upcasting the generated Proxy instance. This can be observed in Figure 4.7.

The `ProxyBase` is constructed to provide `InfotainmentHubSomeIPProxy` access to the Proxy methods from the `capi_runtime`. After the `InfotainmentHubSomeIPProxy` has been constructed, it is initialized, if the initialization succeeds then it is initially upcast to `IInfotainmentHubProxy`, the Proxy interface generated by the code generator, and only after that is it finally upcast to `Any`. Note that these upcasts are done implicitly by Rust. This slight variation ends up making no difference as in both versions if the initialization fails then nothing is returned to the user, and an error message is printed out, this just happens in different places and some additional responsibility is passed to the code generator.

The other part, registration of `Stubs` and creation of `StubAdapters` is slightly more complex. The user is the one responsible for actually creating the instance of its implementation of the Stub and then calls the `register_stub` method of the `Runtime` with additional parameters that allow the specification of which middleware to use. The `Runtime` will then find the

```
fn setup_InfotainmentHubSomeIPProxy( ... ) -> Option<Box<dyn Any>> {
  let base = ProxyBase::new( ... );
  let someip_proxy = InfotainmentHubSomeIPProxy { base, ... };

  if someip_proxy.init() {
    // Upcast to IInfotainmentHubProxy
    let proxy: Arc<dyn IInfotainmentHubProxy> = Arc::new(someip_proxy);
    // Upcast to Box<Any>
    Some(Box::new(proxy))
  } else {
    None
  }
}
```

Figure 4.7: Snippet of Rust proxy setup method for the InfotainmentHub service proxy, details have been omitted for simplification

corresponding middleware implementation of `IFactory` and pass the instance to it. At that point, we are already in middleware dependent code. Then, the respective `StubAdapter` is built with an instance of both the `Stub` and the `Connection` sent to it. The `StubAdapter` then downcasts the `Stub` instance to the Stub interface created by the code generator, for example `IInfotainmentHubStub`, that contains all of the service specifications. With this, the `StubAdapter` is fully built. To finalize the process, the `Stub` receives an instance of this newly built `StubAdapter`, and an instance is also sent to `Connection`. In turn, `Connection` will signal the SOME/IP implementation that the service instance offered by the `Stub` is now available and also registers a message handler for this same specific service instance, thus finalizing the `Stub` registration.

Therefore, from a `capi_runtime` perspective, the `Stub` travels from the higher abstraction level to the lower abstraction level, meaning that in the initial `register_service` method in the `Runtime`, we would already have to receive a common `IStub` to then eventually downcast it again into the specific Stub implementation. This is the approach we have in C++ since through `dynamic_cast` it is possible to do this downcast.

In Rust, since the only construct that can downcast to an arbitrary type is `Any`, the `register_service` call already needs to receive an `Any` object. Since this conversion is not direct, some responsibility is moved to the user. However, the code generator provides a helper method `into_box()` that can be used to simplify this process, as can be seen in Figure 4.8. The method will initially upcast the concrete Stub implementation into the interface the code generator created, in this case `IInfotainmentHub`. Afterwards, it will upcast the object once again into `Any`, which is how when the `StubAdapter` downcasts it, the result will be the `IInfotainmentHub` and not the specific user's Stub implementation. The remainder of the parameters are the ones used to decide which middleware will be associated with this specific service. Naturally, since this implementation only covers SOME/IP, these are redundant but already establish the groundwork for further extensions.

```
let service = hub::Hub::new();
Runtime::get().register_service(
  "local",
  base::InfotainmentHub::get_interface(),
  "hub",
  service.into_box(),
  "Infotainment Hub",
);
```

Figure 4.8: Rust creation of its `Stub` implementation and subsequent registration in the `Runtime`

4.2.2 Data types (de)serialization

A crucial section of the CommonAPI Runtime is the translation between the bytes received from the middleware into their actual data representation, known as serialization and deserialization. It is expected for these procedures to be used several times throughout an application execution, so it should be as performant as possible. After examining the logic of these operations, no flagrant error, redundancy or improvement could be found without resorting to the benchmark of different implementations for multiple inputs, which was not the focus of this work. Hence, there is barely a difference between the Rust and the C++ version other than language-specific constructs or operations. What had to be changed is the location in the full CommonAPI toolchain architecture where these procedures are executed and the code generators role in them. In the reference implementation, the (de)serialization logic relies on very heavily templated code that can accept an arbitrary number of arguments and then process them in order. This makes it possible to have these procedures execute inside the CommonAPI Runtime, and the generators only need to worry about setting the correct template arguments for the methods. Such an approach is currently not possible in Rust as it does not yet support an arbitrary number of template arguments, only a specific number, although the intention to support it exists[3] but it does not seem like it will be anytime soon.

Therefore, it is not possible to have a generic procedure that can support all kinds of input from the generators, which is the approach we see in C++. One of those kinds of generic constructs that can support multiple inputs and outputs can be seen in Figure 4.9. It accepts the user's `Stub` implementation and then receives an arbitrary number of input arguments, output arguments, and their respective deployments. As previously mentioned, deployments configuration parameters that can be defined in the Franca `.fdepl` files that allow the user to customize how the serialization of a type is made. From a developer point of view, these types of constructs are impressive and shows that the author has a deep knowledge of C++ concepts and its limits. However, this is unarguably a nightmare to anyone that seeks to fully comprehend the inner workings of this library, whether out of curiosity or out of necessity.

In contrast, Rust implementation approach follows closely to what is already used throughout the ecosystem. Rust allows the extension of functionality for its standard types through the implementation of user-created interfaces. This is exactly what is used in one of the most widely used

[3] See Issue #376 for the discussion

```
template <
  typename StubClass_,
  template <class...> class In_, class... InArgs_,
  template <class...> class Out_, class... OutArgs_,
  template <class...> class DeplIn_, class... DeplInArgs_,
  template <class...> class DeplOut_, class... DeplOutArgs_>
class MethodWithReplyStubDispatcher<
      StubClass_,
      In_<InArgs_...>,
      Out_<OutArgs_...>,
      DeplIn_<DeplInArgs_...>,
      DeplOut_<DeplOutArgs_...>> : public StubDispatcher<StubClass_> {
  ...
}
```

Figure 4.9: Snippet of one of the C++ classes used to serialize and then deserialize a method call from a remote client

Rust libraries,[4] `serde`, where the interfaces `Serialize` and `Deserialize` are defined, that as the name implies, allow the user to provide a (de)serialization procedure to any type that implements them. Format dependent libraries will then provide the actual implementation for each type, for example, the `serde_json` implements the (de)serialization from and into JSON. For Rust standard types this is enough, but it does not cover user-defined types such as structs or enumerations as the library has no knowledge of these. Once again, the issue can be solved by resorting to Rust powerful macro system. Using `derive` macros, the user can easily make its data type implement the `Serialize` and `Deserialize` interfaces by resorting to the `#[derive(...)]` attribute. This technique is precisely what was also employed in Rust CommonAPI Runtime implementation. For the data types that directly map to Franca data types, the implementation is already present in the library, whereas for the rest of them through the use of these macros an implementation can be provided. How this actually translates into code can be seen in Figure 4.10, unlike in `serde`, only a single derive macro `Serialization` was defined, which implements both the serialization and the deserialization methods.

An addition that had to be made was the specification of the types Franca deployment information, while the deployment information for the user-defined type is passed by the code generators to the (de)serialize functions when they are invoked, it is possible that an inner type also has a specific deployment specification that needs to be used. For this, `derive` macro attributes were used, that allow the specification of additional information to which the macro has access to. The attribute that can be seen in Figure 4.10, `Deployment` associated with the member `inner_struct` is used precisely to specify its deployment. Its value points to the object that contains this information and will then be used by the `derive` macro accordingly. For members that did not have deployment information specified, such as `instance_n` there is no attribute and the (de)serialization uses default values for its type.

[4] According to number of downloads in crates.io

```
#[derive(Serialization)]
pub struct TestStruct {
  #[Deployment = "super::InnerTestStruct_Deployment"]
  pub inner_struct: crate::InnerTestStruct,
  pub instance_n: u8,
  ...
}
```

Figure 4.10: Example Rust struct that gets its (de)serialization implementation from the `derive` macro.

C++ approach to user-defined data types is totally different. As it is impossible to have a template that both accepts an arbitrary struct and can traverse its elements in order, they resorted to creating a generic class for each user data type and using these to actually do the (de)serialization. So the `capicxx-core-runtime` has 3 additional objects, not shown in the initial analysis in Chapter 3, `Struct`, `Enumeration` and `Variant`. The first is simply a struct whose only member is a tuple that contains all the members of the user's struct. The second is also just a struct with the value of the enumeration. Lastly, the `Variant` is a class whose purpose is to provide a type-safe union which appears to be similar to what C++17 `std::variant` can provide, which was only introduced at the end of 2017. Any user-defined data type specified in the Franca `.fidl` files will end up being translated into one of these 3 types, forcing the end-user to use these types and ultimately steepening the learning curve of the CommonAPI toolchain. In this aspect, Rust CommonAPI implementation shines as all user-defined types are translated into Rust standard types, making it much easier for a user to use them.

All of these typing and templating differences between C++ and Rust translate into a different architectural approach for the (de)serialization of data types. In both toolchains the implementation resides in the CommonAPI Runtime library, however, in the Rust toolchain, the code generator has a more direct influence on the correct (de)serialization of types, as opposed to the reference where it is more indirect. The difference resides on where the execution of the (de)serialization procedures is started. For the reference implementation, it is in the CommonAPI Runtime, while in Rust it is in the generated code, as can be observed in Figure 4.11. The method shown in that Figure, `new_frame`, has a single input argument, only, a vector, and no output arguments.

C++ approach is for the code generator to define the serialization ordering through the complex templating of the function `callMethodWithReply` and the order in which the input arguments are passed to the function. The templated arguments are wrapped in a `Deployable` object alongside their deployment specification, which is none in this case, and then handed to the function. If multiple input arguments are required, then the number of arguments of the method grows accordingly. No output argument was specified in this case, but if there were any, the function would receive additional arguments after the `_internalCallStatus` variable which would be overwritten by the function to contain the service response.

Since this variadic number of function arguments is not possible in Rust, it leaves the option to move the (de)serialization outside of the `call_method_with_reply` function, which

```
// C++ implementation
void InfotainmentHubSomeIPProxy::new_frame(
  const AutomotiveTypes::Frame &_frame,
  CommonAPI::CallStatus &_internalCallStatus,
  const CommonAPI::CallInfo *_info)
{
  CommonAPI::Deployable<AutomotiveTypes::Frame,
    AutomotiveTypes_::FrameDeployment_t> deploy_frame(
        _frame,
        static_cast<AutomotiveTypes_::FrameDeployment_t *>(nullptr));

  CommonAPI::SomeIP::ProxyHelper<
    CommonAPI::SomeIP::SerializableArguments<
      CommonAPI::Deployable<
        AutomotiveTypes::Frame,
        AutomotiveTypes_::FrameDeployment_t>>,
    CommonAPI::SomeIP::SerializableArguments<>>::callMethodWithReply(
      *this,
      CommonAPI::SomeIP::method_id_t(0x3e8),
      false,
      false,
      (_info ? _info : &CommonAPI::SomeIP::defaultCallInfo),
      deploy_frame,
      _internalCallStatus);
}

// Rust implementation
fn new_frame(
  &self,
  frame: &crate::Frame,
  info: Option<capi_runtime::CallInfo>
) -> Result<(), capi_runtime::CallStatus_E> {
  let mut bytes = Vec::new();
  ok = frame.serialize(&mut bytes, None, false);
  if ok {
    capi_runtime::someip::call_method_with_reply(
      &self.base,
      0x3e8,
      false,
      info,
      bytes,
    ).map(|_| ())
        } else {
    eprintln!("Failed to serialize 'new_frame' arguments!");
      Err(capi_runtime::common::utils::CallStatus_E::InvalidValue)
    }
}
```

Figure 4.11: Comparison between the C++ and Rust implementation of the proxy for the `InfotainmentHub` service `new_frame` method

is precisely what can be seen in the Figure. It initially creates a new vector and then calls the `serialize` method for each of the input arguments in succession. This is the same method that is provided through the `Serialization` `derive` macro shown above. Only when all arguments have been successfully serialized does it then call the same `call_method_with_reply` procedure, passing the vector used as the output for the serialization which now contains the types respective on-wire representation. When the server replies, the method returns the SOME/IP message, and then the deserialization of its payload will take place, this would be present in the `.map()` section, since there are no output arguments, it is empty.

The semantics through which the service methods output arguments are returned to the user also differs, in C++, function output arguments were preferred, where the function receives references to external variables and then overwrites their value with the response from the service provider, whereas in Rust the preferred approach is the actual declaration of a return type using `Result`, allowing the user to distinguish between the non-error return and the erroneous one through the `CallStatus_E`. Multiple output arguments are grouped into a tuple in the Rust version, in this function the empty tuple `()` is returned on success. Due to a limitation in Rusts type system, Rust only allows the definition of tuples with a maximum of 12 elements, which should be more than enough for the majority of use cases.

4.2.3 Rust CommonAPI limitations and minor differences

There are some additional differences from the Rust version to the reference implementations. These are much minor differences that have little to no impact in the actual logic in the library and are instead corrections to what are probably mistakes, lack of review or planning for features that never came to fruition. A basic one is a necessary dynamic dispatch in someip `Proxy` and `StubAdapter` implementations. Instead of directly storing an instance of `Connection`, this instance is hidden behind an interface named `ProxyConnection`, which `Connection` implements but is not directly used anywhere else. As the exact same functionality could be achieved by removing this indirection, Rust equivalent objects directly store an instance of `Connection`, effectively decrementing the number of dynamic dispatches needed by the library. Then, a less relevant difference, which is a consequence of Rust preference to use static linking instead of dynamic linking, is the single check that the `Runtime` does when trying to find a suitable `Factory` to create Proxies or register Stubs. In the reference implementation, `Factories` can be dynamically linked at that point to provide for a specific required middleware. Since in Rust the preferred approach is to link everything statically, if no suitable `Factory` is found, then we are sure that we cannot use that specific middleware and can just fail without trying to link the appropriate `Factory` dynamically.

In terms of limitations of the Rust implementation, the main one is the unsupported asynchronous calls to the various methods and event fetching capabilities. Finally, since this implementation is only focused on the SOME/IP part of the CommonAPI, the parsing and processing of the CommonAPI `.ini` files is not implemented. It was through these that the reference implementation parsed the information to be able to choose dynamically, which CommonAPI middleware

implementation to use. Nonetheless, its format is straightforward, and it should not be difficult to add this if more middlewares need to be supported.

The last limitation is the inability to use the asynchronous counterparts of the methods created by the code generator. In the reference implementation, a client could call a remote method and pass a callback for when the response is received instead of blocking while waiting for this response. The same is possible in the event access methods. This is something that could certainly be implemented in the Rust version, but ultimately fell into oblivion and ended up not being implemented, although the constructs to have this features are mostly present in the final work.

4.3 Rust SOME/IP implementation

The SOME/IP protocol implementation, named `vsomeip`, is at the heart of the whole Rust CommonAPI toolchain. As the actual middleware used by the applications, nearly all of the useful processing is done inside this implementation. Consequently, it stands to reason that this was the part of the work where the most effort was put, so as to achieve an equivalent or better implementation compared to the reference.

Throughout the development of this work, multiple `vsomeip` versions were released. When planning and beginning the work, version `2.14.16` was the most recent one. Shortly after version `3.1.7.1` was released, with a major version upgrade that signaled some breaking changes. Since it was only the beginning and an up to date implementation was desired, `vsomeip3` was the version that was followed. In the end, the version that ended up fully making it into the Rust implementation was `3.1.16.1`. The high-level architecture remained the same as the reference implementation, previously seen in Figure 3.15. The changes can only be observed in a lower-level view of the individual components, which is what this section will be focused on and, unlike in Chapter 3, a bottom-up analysis will be made.

4.3.1 Endpoints

The lowest `vsomeip` components, the Endpoints, are also where the most changes have been made. This is partly due to the same differences between C++ and Rust shown in the previous section, and also due to a different approach to asynchronous operations. In the reference implementation, we once again have several layers of inheritance that can be observed in Figure 4.12. However, unlike in the CommonAPI implementation, only the upmost interface `Endpoint` is used to actually abstracting which endpoint is being used. The remainder of the abstract classes are simply providing the derived classes with shared behavior between them.

`VirtualServerEndpoint` is an endpoint that does nothing. In the reference `vsomeip` implementation, it is explained that it is used, only, for when the user intends to have the "Service Discovery announce a service that is externally implemented"[5]. Since no use case for such a system could be found, it has simply been left out in Rust implementation, whose architecture can be seen

[5]Present in the User Guide

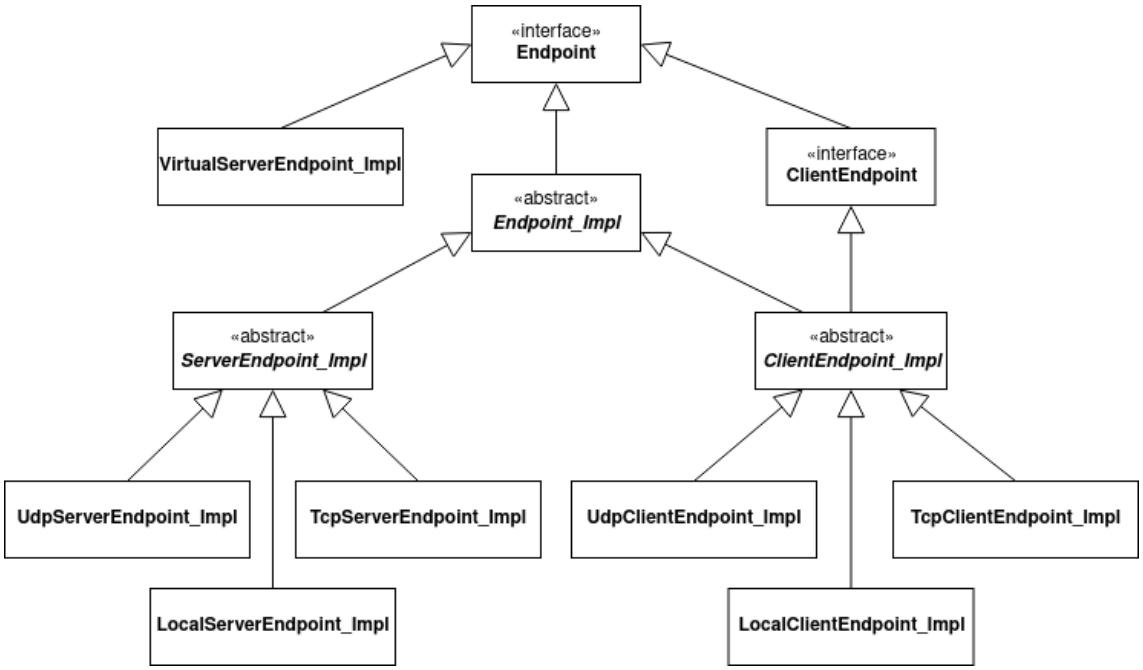


Figure 4.12: Diagram of the C++ `vsomeip` Endpoint objects and their relationships

in Figure 4.13. Unlike what could be expected, the multi-layered inheritance has not disappeared, with 2 middle interfaces still remaining. This is partly true, only. The `IServerEndpoint` and `IClientEndpoint` interfaces are used internally by the Endpoints module, only, to provide the `ServerEndpointBase` and `ClientEndpointBase` with the possibility to abstract over which endpoint it belongs to. However, this is a cost-free abstraction as the base endpoint objects are generic over precisely those two middle interfaces, which means that the functions and function calls get monomorphized at compile time with no additional cost at runtime.

The remaining of the `vsomeip` modules either abstract over the `IEndpoint` or, when possible, directly use an Endpoint object. Rust `IEndpoint` interface contains the same methods as C++ `Endpoint` as well as some additional methods that had to be added as a result of some `dynamic_cast` that the reference implementation resorts to to obtain a specific endpoint implementation and use methods unique to it. As previously explained, that is not possible in Rust, so the solution was to add these methods to `IEndpoint` and have an implementation that throws an error in the endpoint objects that are not supposed to have these. All of the implementations that were present in the C++ middle abstract classes have been moved into either the `ServerEndpointBase` or the `ClientEndpointBase` objects. Then, each of the respective endpoints stores an instance of these objects. This approach has the downside that the implementations present in the `Endpoint_Impl` abstract class are now duplicated in the endpoint base objects, but since it is just a few simple getter and setter methods it seemed pointless to create a whole new object just to share these.

While the architecture is certainly different from the reference implementation approach, it is not the most distinct aspect of Rust implementation. Both the implementations use event-driven,

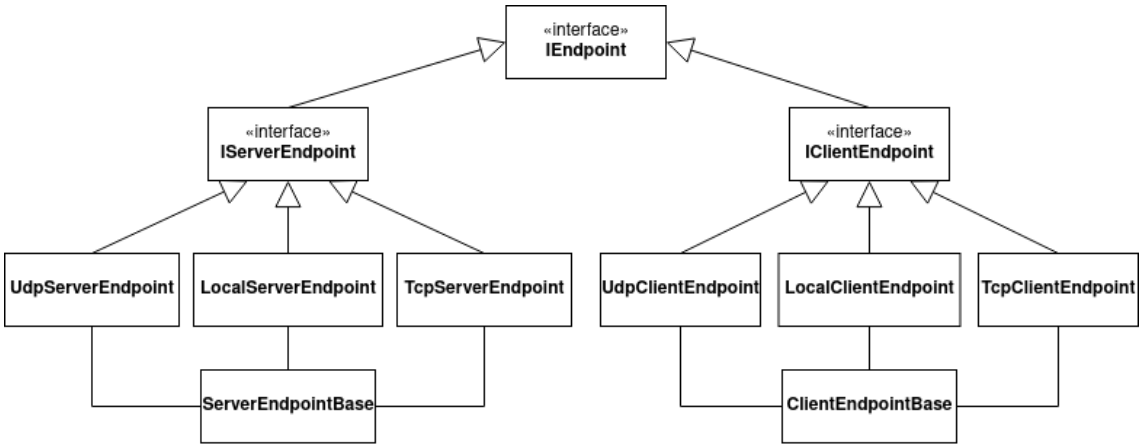


Figure 4.13: Diagram of the Rust `vsomeip` Endpoint objects and its relationships

non-blocking IO operations to write asynchronous applications. Without the use of asynchronous operations, other strategies such as a thread per socket connection would have to be used, which can degrade system performance due to the increased number of context switches required. For the C++ implementation, the `Boost.Asio` library was used whereas in Rust `tokio` was adopted. Despite providing similar capabilities, the API for these libraries and how they are meant to be used by a developer naturally differ. This difference can be quite clearly seen in these components. The reference implementation Endpoints use a design similar to infinite recursion. Their `receive()` method, which reads a frame from the socket, will register a handler for when the frame is successfully read and then exit the function. The `Boost` library will then internally poll the socket waiting for data to become available. Once that is the case, it will then execute the handler on one of its internal threads, managed by the library. These are threads that run in the user space, on top of a kernel thread, and entirely managed in the user space without resorting to system calls. Multiple internal threads can be executed in a single kernel thread. This means that the cost of context switching, creating and destroying one of these internal threads is much lower than an actual kernel thread, as such these types of threads are named lightweight-threads. The low-cost of management and switching between these types of threads makes them ideal for asynchronous applications. Once the socket handler function is executed in one of these threads, it will then process the frame that was just received, and right before it exits, execute the `receive()` method again, effectively creating a loop. Unlike regular loops, this one will potentially span across multiple threads, both lightweight and kernel threads. As such, although conceptually recursion is used, it is a different kind of recursion that does not entail the increase of the method call stack. To ensure that no synchronization problems occur, it is necessary to use a mutex to synchronize access to the socket and access to the receive and write buffers.

Rust `tokio` approach is similar from a library perspective, where the use of lightweight-threads also prevails but differs in the API and how it is supposed to be used by the developers. Contrary to `Boost`, `tokio` gives the developer the ability to create and delete these lightweight-threads that then get executed by the `tokio` Runtime. These threads can communicate with each

```
fn start_socket_loop(
  self: Arc<Self>,
  mut socket: TcpStream,
  mut data_rx: mpsc::Receiver<Arc<[u8]>>,
  mut stop_rx: watch::Receiver<Service_T>,
) {
  tokio::task::spawn(async move {
    loop {
      tokio::select! {
        data = data_rx.recv() => { ... }, // Send message
        msg = socket.recv() => { ... },   // Process message
        _ = stop_rx.recv() => { ... },    // Stop loop
      }
    }
  });
}
```

Figure 4.14: Rust-based pseudocode of an `Endpoint` socket loop

other either via synchronization primitives, or, the preferred approach, through message passing using a queue where a thread can write to it, and another read from it. This is exactly the design that was chosen for the Rust implementation, as shown in Figure 4.14. Instead of having the type of "recursion" seen in the reference implementation, for each socket a single lightweight-thread is spawned which is responsible for both reading and writing to the socket, done by the `tokio::task::spawn()` method. Inside this thread, there is no need for any synchronization primitive as it is the sole thread with access to the socket and its read and write buffers. For communication with the remaining code, a queue is used. In this case, only a single queue is required to send the messages. The thread will read the message from the queue and then send it to the socket. When a message is received, it simply starts executing the respective frame processing procedures, which means that it will not be sending or receiving messages in the meantime. The procedure can wait concurrently for different events due to the `tokio::select()` method, which has a similar behavior to the Linux `poll()` and `epoll()` methods.

With this type of approach, the amount of synchronization primitives needed for each endpoint is significantly reduced, which, in turn, also removes the overhead from their operations. As an example, the `LocalClientEndpoint_Impl` from the C++ implementation has up to 5 mutexes that it needs to use throughout its different procedures, although one of them is only seldom used. Whereas Rust equivalent `LocalClientEndpoint` needs to use 1 mutex, only, for exactly the same operations. This sharp difference is only possible due to the message passing architecture used in Rust which allows the threads to execute independently from one another and synchronize with one another through message passing, effectively avoiding shared state and thus the need for synchronization primitives. The same design is seen throughout all the different kinds of Endpoints shown in Figure 4.13, meaning that ultimately quite some synchronization primitives have been made redundant and consequently their runtime cost eliminated.

Another relevant yet simple contrasting aspect that the Rust Endpoints have compared to the reference Endpoints is their monomorphization. The Endpoints need to store an instance to both

its `EndpointManager` and the `RoutingManager`. In the reference implementation, these instances are stored through interfaces, `endpoint_host` and `routing_host`, respectively. This means that each time a method from them needs to be called dynamic dispatching is required, effectively adding to the total latency of the library. While it is true that only a few method calls are made it undeniably adds additional overhead. The same situation is handled differently in Rust, by making all the Endpoints have a templated argument for both the `EndpointManager` and the `RoutingManager`, the compiler can then monomorphize these function calls and remove the cost of dynamically dispatching these at runtime, further contributing to the latency reduction.

A feature present in the reference implementation that did not make it into the Rust implementation is the buffering of outgoing messages. The C++ version implements what is named "train buffers" or the nPDU feature. These are buffers that have a departure time and will only be sent through the socket after a certain period of time. Until then, when the library intends to send a message, it will simply be copied to the trains buffer and only be sent, alongside other messages, once the departure time has been reached. There is no AUTOSAR requirement for the buffering of messages in either the SOME/IP or the SOME/IP-SD specification. In fact, there is even the mention in section 4.2.1.2 of [7], that "in order to reduce latency and reaction time", when using TCP, Nagle's algorithm should be turned off. So clearly a lower latency should be something to strive for. According to the documentation of the GENIVI SOME/IP implementation, this was implemented in order to lower the network load in exchange for latency. The exact timings for this can be configured. However, even when setting no retention time and having the trains depart immediately, it already causes additional latency simply because it still will copy the whole message into the train buffer. This feature was completely removed in the Rust implementation. The library makes no kind of buffering, as soon as messages are received via the queue, the socket thread will immediately write it to the socket. Using this approach, the latency is significantly reduced, first due to the fact much less code is executed between the message reaches the `Endpoints` until it is actually sent to the socket, but, most importantly, no copy of the outgoing messages is made. The copy overhead may be negligible for small messages, but it can quickly add up to a significant amount.

4.3.1.1 Endpoint Managers

The `EndpointManagers`, responsible for creating and managing `Endpoints`, have a fairly simple role in the overall architecture. As such, both implementations are very similar, if not equal, in terms of its algorithms. Architecture wise, as already observed in other locations, it needs to be slightly changed to the familiar approach of moving the shared implementations to an object and have an instance of it where this implementation is required as seen in Figure 4.15. Other than that, an additional manager was created, `ProxyEndpointManager`. This manager functions simply as a wrapper around the `EndpointManagerBase` and redirects all of the function calls to it. The reason it was created was, once again, to allow the monomorphization of the `EndpointManager` instances in the `RoutingManagers`. Even more monomorphizations were made in the `IEndpointManager` methods that create and manage local endpoints. For these

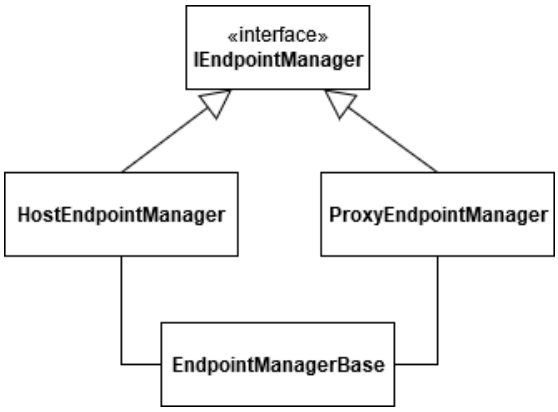


Figure 4.15: Diagram of the Rust `vsomeip` EndpointManager objects and its relationships

methods, we know that by design, they will be returning either a `LocalClientEndpoint` or a `LocalServerEndpoint`. As such, instead of abstracting them behind the `IEndpoint` interface, we can use them directly both here and in the other `vsomeip` components that execute these methods. The remaining of the client and server endpoints need to be abstracted by `IEndpoint` as they can either be the TCP or UDP implementation.

4.3.2 Routing Managers

The Routing Managers represent the central component that manages most of the SOME/IP implementation logic. As explained before, there are 2 types of Routing Managers, the Host and the Proxy, the former is used with the Host `vsomeip` applications and the latter with the remaining applications in that system. At runtime we cannot know which of those 2 types the application is going to be, hence why the Routing Managers need to be abstracted through the `RoutingManager` interface in the same manner seen in Figure 3.15. This of course still holds true in the Rust implementation with the interface `IRoutingManager` being used to dynamically dispatch these.

After that necessary abstraction, by design no further, abstraction should be required apart from the `Endpoints`. So all of the helper objects of the Routing Managers will belong to either the `HostRoutingManager` or the `ProxyRoutingManager`. This makes these objects the perfect candidates for being templated objects generic over a `IRoutingManager` and as such subjected to the compiler monomorphization. Despite this, in the C++ implementation, dynamic dispatch is still used in many different places. In Figure 4.16, an overview of Rust implementation objects, their relationships and, in bold, locations where static dispatch was used instead of dynamic dispatch is shown. The numbers represent the storing of an instance of the object, for example, the `LocalServerEndpoints` shown contain an instance of `RoutingManagerHost`, but the `RoutingManagerHost` does not have direct access to this instance of `LocalServerEndpoint`, although it could still access it through the `EndpointManager`.

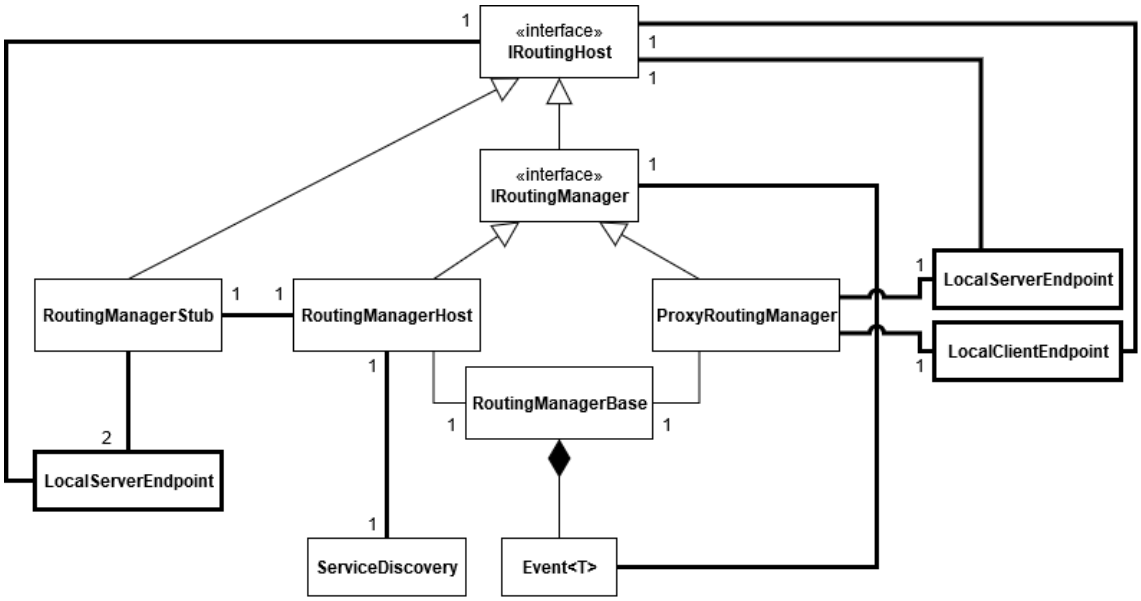


Figure 4.16: Diagram of the Rust `vsomeip` RoutingManager objects and its relationships, bold represents relationships that use static dispatch instead of dynamic dispatch

Again, the Routing Managers follow the already explored approach of storing the shared implementation of `IRoutingManager` in `RoutingManagerBase` and then redirecting calls to it. Note that the relationship between `Event` and `IRoutingManager` is not implying dynamic dispatch. As represented by the template argument in `Event`, since each event can only be associated with a single `RoutingManager` and each `RoutingManager` only stores events associated with it, this relationship will be monomorphized at compile time. This `RoutingManager` instance will be used by `Event` each time that it needs to send a notification to a remote client, so only applications that rely on event-based communication are subjected to the dynamic dispatch overhead. The component responsible for the SOME/IP-SD protocol also does not require any kind of dynamic dispatch. By design, the `RoutingManagerHost` is the only Routing Manager that can have Service Discovery enabled, and so `ServiceDiscovery` necessarily is owned by `RoutingManagerHost`. Hence, in this case, monomorphization is not even required, it simply stores an instance of `RoutingManagerHost`.

The same situation applies to the `RoutingManagerStub` where by design, it belongs to the Host, only, and can also directly store an instance of it. Both of these objects rely heavily upon accessing this `RoutingManagerHost` instance. In the `RoutingManagerStub` case, the instance is used once per message that needs to be sent to local clients and, depending on the type of response received, at least once for each response from local applications. The exchange of these local messages is more frequent in the setup of the system, where all of the local applications are required to register themselves with the Host application which, in turn, informs the remaining applications of the newly registered application. After this initial setup, during the regular execution of the application, any local communication is composed mostly of messages sent by the `RoutingManagerHost` as a mean to propagate some change in status,

methods or events received from remote applications and possible subsequent responses from the local applications. In the `RoutingManagerStub`, the fact that it directly stores instances of `LocalServerEndpoints` instead of the dynamically dispatched `IRoutingManager` makes no difference as it never needs to access these members, they are only used to receive messages from the local applications. When a new local message is received, these Endpoints then send it to the `RoutingManagerStub` to be processed. It is in this direction that the deletion of the dynamically dispatched types can be felt, as the Endpoints are templated over the `IRoutingHost` interface, instead of dynamically dispatching its implementation. This interface also exists in C++, it was simply not shown in previous diagrams for simplicity purposes. This templating applies to every single endpoint of the program, and so its consequences can be felt everywhere in `vsomeip`.

Finally, on the other side of the Figure, the `ProxyRoutingManager` is located. Here a situation identical to `RoutingManagerStub` can be seen, with the storage of the specific endpoint instances instead of an abstraction over the `IEndpoint` interface and the endpoints having direct access to `ProxyRoutingManager` through their template argument. In this object, owned by all the local applications which are not a Host application, the other side of local communication is present. Through its instance of `LocalClientEndpoint` it sends all of the local messages to the Host application.

4.3.3 Service Discovery

The `ServiceDiscovery` component is where most of the SOME/IP Service Discovery protocol is implemented. As was shown, it stores an instance to the `RoutingManagerHost` that it uses to aid the protocol implementation. Once more, this is a specific instance with no need to be dynamically dispatched, unlike the reference implementation. The Service Discovery protocol, at its core, is relatively simple, services announce their existence by sending OFFER messages and attempt to find other services in the network by sending FIND messages. These are sent through multicast, so every device in the network receives them, OFFER messages keep being sent throughout the application execution, while FIND messages are only sent at startup to find services quickly.

The Service Discovery implementation needs to follow a state machine, as defined in the AUTOSAR specification [2], that can be seen in Figure 4.17. In it, two different branches are shown, first, the one related to the OFFER messages and the one for the FIND. Both of these follow the same behavior at startup. The Initial Wait phase will last for a random delay within an interval specified by the user, while gathering all of the services that the application is offering, in OFFER messages, and the ones that the application is requesting, in FIND messages. After this initial delay, it will send these messages for the first time, thus exiting the Initial Wait phase and moving onto the Repetition Phase. For this phase, the user specifies a base delay (BASE_DELAY) and a maximum number of repetitions. Immediately after entering it, the application shall wait for $2^0 * BASE\_DELAY$ milliseconds, before sending resending the messages gathered in the initial phase. Afterwards, it waits for $2^1 * BASE\_DELAY$ and sends these messages again, with this process repeating until the maximum number of repetitions has been reached, marking the end of

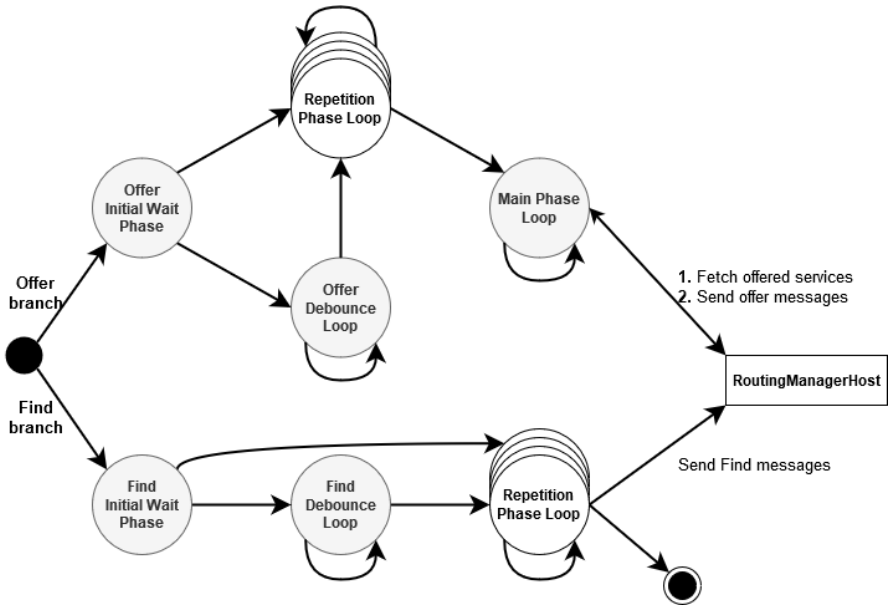


Figure 4.17: State machine of the SOME/IP Service Discovery protocol startup behavior

the Repetition phase. This also marks the end of the state machine FIND branch, and no more FIND messages will be sent throughout the application execution. On the other hand, the OFFER messages enter the Main phase where they wait a specified amount of type and then get sent again. This process repeats endlessly throughout all of the application execution, so OFFER messages keep being sent into the network. For requests or offers that are made after the Initial Wait phase has finished, an additional Debounce exists, where, similarly to the Initial Wait phase, it gathers requests and offers for a certain amount of time (DEBOUNCE_PERIOD), if any is received within that time frame it sends the messages and then moves them into the Repetition phase.

Rust implementation of this state machine utilizes lightweight-threads and message passing between them. The grayed out states in Figure 4.17 represent the threads that are spawned immediately on application startup. However, note that the Initial Wait phase and the Debounce phase are executed in the same thread. Both of these phases gather the services in a collection, and after the first messages are sent, a new Repetitions phase thread is created with this collection of services being passed to it. This is made without copying all of the collected services as the whole collection is passed to the Repetitions thread while a new empty collection is left in its place in the Debounce thread. That Repetitions phase thread will only be responsible for the services passed to it and once that phase is over will terminate. Thus, an arbitrary number of Repetition phase threads may exist, so Figure 4.17 represents that state as a stack of states. The Main phase thread will be constantly fetching from the `RoutingManagerHost` the application offered services, creating the OFFER messages with them and finally sending them to the network.

All of the threads from a branch, except that of the Main phase, share the receiving end of a queue used to signal when to stop offering or requesting a specific service. The Main phase thread does not require one as it is directly reading from the Routing Manager storage from which the services are deleted when they stop being offered. This approach followed in the Rust im-

```
void debounce_timer_expired() {
  collected_offers_mutex.lock();
  new_offers = collected_offers;
  collected_offers.clear();
  collected_offers_mutex.unlock();

  send_offers(new_offers); // Sends the offers for the first time

  new_timer = std::make_shared<steady_timer>();
  timers_mutex.lock();
  timers[new_timer ] = new_offers;
  timers_mutex.unlock();

  // Sets timer to wait and call handler afterwards in different thread
  new_timer.setup(500, |new_timer| {
    timers_mutex.lock();
    offers = timers[new_timer];
    send_offers(offers);
    new_timer.setup( ... );
    timers_mutex.unlock();
  });
}
```

Figure 4.18: Pseudocode based on C++ collection of offers during the offer debounce state

plementation is different from the one in the reference implementation, where `boost` timers are used instead of lightweight-threads. An overview of this process in the reference implementation is shown in Figure 4.18.

For the gathering of services, a shared collection, `collected_offers`, is used that needs to be synchronized via a designated mutex. Once the gathering phase is over, a new `boost` timer is created, which will be handling the Repetition phase. The association between this new timer and the gathered services is made through another shared map that links a timer to the services gathered in `collected_offers`. Unlike Rust implementation, the services in `collected_offers` are all copied into that shared map and afterwards the `collected_offers` is cleared. As is usual, this map is also synchronized via a mutex. When the timer expires, it locks the map mutex and only releases it after having sent the messages. Since all timers share this map, they are dependent in one another as only a single timer expiration handlers can be executed at any given time. Whereas in the Rust implementation, they are independent from one another and can run concurrently without any issues. Once the Repetition phase is over, C++ follows the same procedure as Rust, setting a flag in the services that signals that they are now in the Main phase.

The procedure described above marks the mandatory behavior that a SOME/IP application offering and requesting services needs to follow and is the base for the remaining exchanges between applications. Applications receiving these messages will then react accordingly, as was explained in Section 3.4.2.

```
{
  ...
  "service-discovery" : {
    "enable" : "true",
    "multicast" : "224.244.224.245",
    "port" : "30490",
    "protocol" : "udp",
    "initial_delay_min" : "10",
    "initial_delay_max" : "100",
    "repetitions_base_delay" : "200",
    "repetitions_max" : "3",
    "ttl" : "3",
    "cyclic_offer_delay" : "2000"
  }
}
```

Figure 4.19: Service Discovery related options of a standard `vsomeip` configuration file

4.3.3.1 Service Discovery configuration

The Service Discovery protocol behavior regarding its various states and respective timeouts is expected to be configured to the user's preference. This can be done via the `vsomeip` JSON configuration files, already shown in Chapter 3. The specification for these files includes a section for the Service Discovery that allows the user to enable Service Discovery and configure some of its options. Without enabling it, it is impossible to have any kind of event-related communication between applications, even if they are being executed in the same system. The most common options are shown in Figure 4.19. At the least, it is important that the `multicast`, `port` and `protocol` options match between SOME/IP applications. Otherwise, they will not be able to discover each other as the SOME/IP Service Discovery multicast communication will be associated with that specific address and protocol. The remaining of the options could, in theory, be different between applications as they affect mainly the timeouts of the offering and request state machine. `initial_delay_min` and `initial_delay_max` correspond to the minimum and maximum range used to generate the INITIAL_DELAY value for the Initial Wait phase. `ttl`, is a value represented in seconds representing the lifetime of the application offered services, it is used to detect abrupt remote application termination. The remainder of the timeout values are represented in milliseconds.

4.3.4 Runtime and Application

The `Runtime` and the `Application` represent the uppermost layer of `vsomeip`, meaning that they are the objects available to be used by external applications. The former is stored as a singleton class in the C++ implementation, accessible anywhere in both the user code and the `vsomeip` library. In practice, it is used in the setup of the user application, only, and nowhere else. Inside `vsomeip` it simply is not used, except for some static methods it provides which do not require an instance of `Runtime`. As such, it felt unnecessary to make it a singleton object. Instead, it is a

regular object that can be created and then used to create the `Applications`, since these are not dependent on the existence of a `Runtime` instance it can be deleted afterwards. The only practical use case for the existence of the `Runtime` is for when multiple `Applications` are assigned the same name, where the `Runtime` will then append a unique identifier to the applications name to make it unique within the system. Such a situation should be extremely unlikely to happen, except by user mistake, and the consequences that either approach may have are fairly similar and related to the SOME/IP identifier that gets assigned to the application.

In any case, since it barely makes a difference the `Runtime` was kept in Rust SOME/IP implementation, also to maintain the API similarity between the Rust and C++ implementations. Precisely for this reason, Rust `Application` public API is also kept exactly the same as the reference implementation. There had to be some slight adaptations since default arguments and overloaded methods are used in the reference, and neither of these is possible in Rust. For the former case, Rust version simply forces the user to explicitly specify all parameters, while on the latter the name of the overloaded method was adjusted to be unique and better match its behavior. Arguably doing so is a better approach as it is clearer exactly what is being passed and executed in both cases.

As `vsomeip` resorts to asynchronous operations, it also stands to reason that the interaction between the library and the user is also done asynchronously. Apart from the setting up of the application where the registration of services and events is made, all of the interactions are made through the execution of handlers that the user-defined. With this in mind, it makes sense that the main focus of `Application`, aside from relaying calls to the Routing Managers, is precisely the processing and dispatching of all of these handlers. In both implementations, this is done through a loop entered once the `Application start()` method is called, which will only return once this loop has finished. In it, the function keeps waiting for new handlers to be available to execute them.

For the reference implementation, this is again done through shared state and synchronization primitives, as seen in Figure 4.20, where a simplified version of this handler dispatcher is shown. Handlers are received through a double-ended queue, accessed through the `get_next_handler()` call in the Figure, and, once again, a mutex that synchronizes its access (`handlers_mutex`). Through the use of a condition variable (`dispatcher_condition`) the loop blocks until it is notified that it has received a new handler to process. Once this happens it will enter the `else` branch of `main_dispatch()` where it first checks if this dispatcher is allowed to dispatch handlers and then fetches the newly received handler. Before beginning the execution of the handler, it unlocks the `handlers` mutex and creates a new timeout. The timeout is created to identify when the dispatching has blocked while executing the user's registered handler. This will only happen if the user's handler takes too long to execute, the default maximum allowed dispatch time is 100 milliseconds.

If the handler finishes the execution within the time window, then the timeout is canceled and this thread resumes dispatching as usual. If the handler does not finish within the time window, then the dispatcher thread identifier is inserted in `running_dispatchers` collection, just

```
void main_dispatch() {
  (its_id, its_lock) = (this_thread::get_id(), handlers_mutex.lock());
  while (is_dispatching) {
    if (!is_active_dispatcher(its_id) || handlers.empty()) {
      dispatcher_condition.wait(its_lock);
    } else {
      sync_handler its_handler;
      while (is_dispatching && is_active_dispatcher(its_id)
             && its_handler = get_next_handler()) {
        its_lock.unlock();
        steady_timer new_timer(100, timeout_callback);

        // Busy waits `dispatcher_mutex` then insert
        running_dispatchers.insert(its_id);

        (its_handler)(); // Invoke handler

        new_timer.cancel();

        // Busy waits `dispatcher_mutex` then erase
        running_dispatchers.erase(its_id);

        its_lock.lock();

    ...
}
```

Figure 4.20: C++ based pseudocode representation of `Application` handler dispatch method from the reference implementation. Parts of the code have been omitted for simplicity sake.

before it executes the handler. Once again, this collection is protected by a mutex, `dispatcher_mutex`, which the dispatcher will busy loop until it is able to lock the mutex to then access the `running_dispatchers` collection. Since the handler did not return, the thread identifier will remain in that collection when the `timeout_callback` is executed. This callback will then use the `running_dispatchers` collection to check if the original dispatcher has blocked. If that is the case, it will create a new dispatcher thread that will take over as the `Application` handler dispatcher. This secondary dispatcher will remain the handler dispatcher for as long as the original dispatcher does not return and thus does not remove its thread identifier from the `running_dispatchers` collection. When that happens, the secondary dispatcher will exit, and notify the original dispatcher that it can resume dispatching through the `dispatcher_condition` variable.

For the Rust implementation, these dispatcher threads are also present, but how they coordinate between one another is very different. The main function for the dispatching can be seen in Figure 4.21, which also has some omitted parts for simplicity sake. As usual, locking mechanisms are replaced by message passing and also atomic variables. The pillar of the Rust implementation are the so called batons, that represent whether a dispatcher thread is allowed to dispatch. A baton is represented by an atomic boolean, as seen by variable `baton` that starts with the `BATON_MINE`

```
fn main_dispatch(&self) {
  let baton = Arc::new(AtomicBool::new(BATON_MINE));
  let (baton_tx, baton_rx) = crossbeam_channel::bounded(0);
  let (wait_tx, wait_rx) = mpsc::channel(1);
  spawn(wait_dispatch(baton_tx, baton, wait_rx));

  while let Ok(its_handler) = self.handlers_rx.recv() {
    if use_handler(&its_handler) {

      baton.store(BATON_PASS);
      wait_tx.try_send(());
      (its_handler.handler.as_ref())(); // Invoke handler
      let i_own_baton = baton.compare_exchange(BATON_PASS, BATON_MINE);

      if i_own_baton {
        wait_tx.try_send(()); // Try to cancel timeout
      } else if !baton_rx.recv() { // Wait until baton is transmitted back
        break; // Dispatching was stopped, exit
      }
    }
  }
}
```

Figure 4.21: Rust based pseudocode of the implemented `Application` handler dispatch. Parts of the code have been omitted for simplicity sake.

state, which represents that the thread is allowed to dispatch. There are 2 message passing components created, the `baton` message queues and the `wait` message queues. The former is a 0 sized channel that functions as a rendezvous channel, where a message can only be sent if the sender and receiver are accessing the channel at the same time, otherwise the sending or receiving fails or is blocking until the other party arrives. The latter is a regular 1 sized channel that is not used to actually send any data, only to notify the other end of some event.

Then, the blocked dispatcher detection thread `wait_dispatch` is spawned, which will have access to `baton`, the sending end of the baton channel and the receiving end of the wait channel. The dispatcher then enters the dispatching loop, where it will continuously read from the `handlers_rx` channel, which is where the rest of `Application` components will be sending handlers to be executed. After checking if the handler is valid, it will initiate the handler dispatching section of the function. The first operation to do is change the `baton` state to `BATON_PASS`, that represents the thread will start dispatch and so, if it takes too long, another thread should get the baton and take over the dispatching. A notification is sent to the wait channel, to signal the timeout detector thread it should initiate a timeout, and then finally the handler is executed. Immediately after the handler finishes execution, it atomically fetches, tests and updated the atomic boolean value. It tests if the current value is `BATON_PASS` and if that is the case it replaces the current value by `BATON_MINE` atomically. The `i_own_baton` boolean will only be true if this exchange was successful, thus if the previous value was `BATON_PASS`. On success, it signals again the timeout detector thread to cancel the timeout and then resumes dispatching. If it fails,

meaning another thread has stored `BATON_MINE` already and thus is dispatching, the main dispatcher will then wait for the baton to be sent back via `baton_rx`. The value that is sent in this baton channel is a boolean, but it is independent of the actual `baton` value, it only serves to signal if the dispatching should continue or not.

```
fn wait_dispatch(&self, baton_tx, baton, wait_rx) {
  use tokio::time::timeout;

  while wait_rx.recv().is_some() {
    if baton.load() == BATON_PASS {
      match timeout(100, wait_rx.recv()) {
        Err(_) if baton.compare_exchange(BATON_PASS, BATON_MINE) => {
          spawn(secondary_dispatch(baton_tx));
        },
        _ => (), // Dispatch finished before timeout
      }
    }
  }
}
```

Figure 4.22: Rust based pseudocode of the implemented `Application` dispatch timeout detection. Parts of the code have been omitted for simplicity sake.

The dispatch timeout thread is very simple with only a few lines of code, as shown in Figure 4.22. It continuously reads from the wait channel, until it receives a new value, meaning it should initiate the timeout measurement. Immediately after, it checks if a dispatching is indeed occurring by testing if the `baton` value is `BATON_PASS`. If that is the case, it will create a new timeout object, which will timeout if the operation `wait_rx.recv()` does not complete within the 100 milliseconds. Should that operation complete, nothing else is required and it will begin waiting for another wait notification. When the operation does timeout, it enters the `Err(_)` branch where it will immediately do another atomic fetch and update, trying to store the `BATON_MINE` state if the previous state was still `BATON_PASS`. If that succeeds then it means the main dispatcher is still executing a handler and thus should be considered blocked.

A new secondary dispatcher will then be spawned, that will receive the sender part of the baton channel. The `secondary_dispatcher()` function is mostly equal to the `main_dispatch()`, and will also create his own `baton` atomic boolean and use the same `wait_dispatch()` for blocking detection. Its only difference is that after every successful handler dispatch, it will attempt to hand control back to the `main_dispatch()` by trying to send a value to `baton_tx`. If it succeeds, it means the main dispatcher has now finished executing the handler and is ready to continue dispatching, so this secondary dispatch will exit afterwards. When it fails, the secondary dispatcher will keep dispatching and repeating the same behavior. Through this approach, there is no need to use mutexes or conditional variables. Also, there is barely any code before or after the handler is executed. In the reference implementation, for each handler, a new timer needs to be created and setup, alongside the locking of the `dispatcher_mutex` and accesses to the `running_dispatchers`. All of these operations have a cost much higher than modifying an

atomic variable and attempting to send a value to a channel. For the Rust implementation, in the majority of the cases, where the handler will not take very long to execute, there may even not be a need for `wait_dispatch()` to create a new timeout.

4.3.5 Implementation limitations

GENIVI SOME/IP implementation is a ful-fledged implementation compliant with all of AUTOSAR SOME/IP specification and also adds implementation-specific features. Since it has been in development for years now, it is already a very mature project tried and tested several times. That is the reason why the Rust implementation followed closely with the GENIVI implementation. Nonetheless, not every single feature of GENIVI `vsomeip` transpired into the Rust implementation. The most relevant feature that was left out was the SOME/IP Transport Protocol (SOME/IP-TP). This is another SOME/IP protocol, that contains a dedicated AUTOSAR specification document, for transportation of payloads larger than the maximum allowed by the network transport protocol used. These will get segmented into different packets and sent to the service. The service will then handle the reassembly of these packets. Since the implementation of this protocol was not within the scope of this work, it was left out. Another feature that has not been implemented is `vsomeip` "Security" module, responsible for ensuring that the local messages received from unknown devices are not processed. As this is not part of AUTOSAR SOME/IP specification, it was also left out of this implementation. Even so, this feature does not drastically change the logic already present in the Rust implementation and the places where this module would be executed are already present in the code, they simply do nothing. Lastly, another feature of `vsomeip` not present in the Rust implementation is the possibility to restart an application. In `vsomeip` it is possible to start an application, stop it, and then start it again, effectively restarting it. Implementing this in the Rust implementation would require some non-trivial changes to the source code with, arguably, little return. Also, since most use cases for a SOME/IP application do not require it to be able to restart at runtime, this capability was left out.

4.4 Summary

This chapter focused on the architectural aspect of the Rust CommonAPI toolchain implementation and its differences compared to GENIVI reference implementation. The toolchain entry point, the code generators, is where an adaptation of the reference generators was made to produce Rust code instead of C++. In there, we presented a comparison between the implementations with some criticisms of the original implementation, how these were approached in the Rust implementation and the reasoning behind these changes. Moving down the abstraction layer, the actual CommonAPI library was explored. Focus was given to how the Rust implementation managed to remain middleware independent, much like the reference implementation, in spite of Rust less flexible typing system. Rust approach to the (de)serialization of Franca data types was also compared to the C++ procedures with the logic remaining the same, but the methodology having arguably become much simpler to understand. Lastly, at the lowest level of the toolchain, light was also shed

into the design of Rust implementation of the SOME/IP and SOME/IP Service Discovery protocol, building on top of what was shown in the previous chapter. A closer look at its components was taken, for each of these how they were designed and the implications that Rust modifications have when compared to the reference implementation was studied, including the Service Discovery section. In the next chapter, the verification of Rust implementation and further analytical analysis will be made between both implementations.

Chapter 5

Implementation validation

In pursuance of ensuring that the Rust implementation of this work is correct and adheres to its requirements, a series of integration tests were developed. Considering that the SOME/IP implementation is the main focus of this work, it stands to reason that it is the most tested component. The remainder of the tools, the code generator, and the CommonAPI implementation, only contain more rudimentary tests not worthy of mention. Nonetheless, some of the tests that will be explored in this chapter make use of such tools and thus indirectly assert their correctness.

5.1 Toolchain integration and interoperability

To put all of the tools together and actually verify that they integrate with one another, the example system presented in Chapter 3 was developed. This naturally started with the `.fidl` and `.fdepl` files shown in Appendix B.1 and B.2, followed by the code generation based on these files. On top of the Infotainment Hub service application, a simple command-line interface was developed to mimic an actual head unit, allowing the user to view and interact with the vehicle status, shown in Figure 5.1.

The interface contains a top bar that shows the availability of each of the remainder of the network services, the Front Wheels, Back Wheels, Seats and Camera respectively. When the service is available, a green square is shown next to it otherwise, it turns red. At the left of the top bar, the process identifier is shown to allow shutting down the application abnormally. In the main display area, below the top bar, the vehicle's overall status is shown. On the left, the vehicle's current speed is shown and in the middle a diagram of the car, where the front wheels are located on the top and the back wheels on the bottom. Under normal circumstances, all of the wheels are white, like in Figure 5.1. However, if the tyre pressure drops below 1.9 psi, the wheel would begin to blink, and if there is any damage to the tyre, it will turn red. Inside this diagram, 5 squares represent the 5 seats of the vehicle. If the seatbelts are on, then the squares are green, and if they are off, they are red. To the right of the vehicle diagram, there is a 5x5 square of pixels representing the current image feed from the camera. Finally, at the bottom of the interface, there is the menu which the user can browse using the directional keys. The currently selected option is underlined,

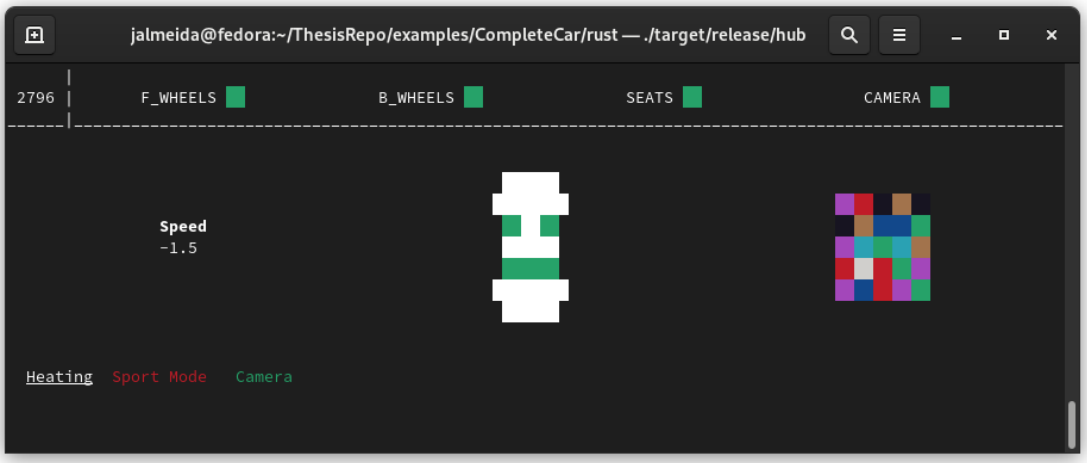


Figure 5.1: Screenshot of the main screen of the Infotainment Hub command-line interface

both the "Sport Mode" and the "Camera" option are simply toggles to turn either the Wheels sport mode or the Camera feed on or off, making the text green and red respectively. Selecting the first option, "Heating", changes the menu and then the user can update the heating settings of each seat individually, as seen in Figure 5.2. In front of the name of each seat, the current status of the heating is shown. When the user is changing the status, the colors are inverted, and the user can scroll through the different options until he finishes the selection, as seen in the Figure in the "Middle" option.

The system will be simulating actions throughout its execution that will be reported back to the Infotainment Hub and shown to the user. As previously explained, the 2 Wheels services emulate changes in speed by following a mathematical function causing the speed to fluctuate between positive and negative values regularly. Both Wheel services are sending their own simulated speed back to Infotainment Hub. Since they follow the same formula and timings, the speed that is shown to the user is only the one from the front Wheels. No user action can influence the speed value as it is a `readonly` attribute of the service. The pressure of the tyres initially starts at 2.1 psi and decreases by 0.1 every 5 seconds. Starting from 15 seconds after the Wheels service application has started, every 30 seconds, a random tyre in each Wheel service will puncture. All of the damage to the tyres and pressure is reset to the initial values every 30 seconds of execution. The Wheels "Sport Mode" is a flag that can only be toggled by the user. The Seats service will remain unchanged for the first 5 seconds of its execution. After that initial statis, the status of the seatbelt of a random seat will be toggled after an interval ranging from 2 to 5 seconds. By default,

Figure 5.2: Screenshot of the heating selection menu of the command-line interface while the user is changing the heating of the back middle seat

the Camera service is not transmitting its frames to the Infotainment Hub, and will only do so when the vehicle's speed is negative. At this point, the Infotainment Hub activates the Camera, which starts sending frames each half a second. Even without user interaction, the Camera will be turned on and off in reaction to the Wheels speed value. Nonetheless, the user can turn the Camera on or off manually.

To shutdown the system and allow all services to exit gracefully, when in the main menu, if the user presses the `Escape` key, the application will broadcast its `shutdown` notification. The broadcast will be received by all other applications that will initiate the `vsomeip` shutdown procedure. At the same time, the Infotainment Hub waits some milliseconds before shutting down after sending the notification.

The process identifier of the Infotainment Hub is shown to allow us to abnormally terminate the process, for example, through a `KILL` signal. Due to the abnormal shutdown, the application will not send the broadcast to the other applications, and they would continue execution and keep the vehicle's state stored. After restarting the Infotainment Hub application, it will re-establish the connection to all the other services. Then, the services send back their stored states and the Infotainment Hub service updates the information shown to the user. This procedure proves that the vehicle's system is genuinely being updated in reaction to the user's interaction.

In this system, it is also possible to change an application to their C++ equivalent. An example would be to terminate the Rust Wheels service application and replace it with its C++ equivalent. The Infotainment Hub would recognize that a new provider of the Wheels service is available and would initiate communication with it just as if it were a Rust program. This is a testament to the Rust implementation interoperability with the reference implementation. Of course, this fact is greatly aided by the SOA that the SOME/IP protocol follows.

5.2 SOME/IP implementation validation

The truly crucial component both for the example system and any other automotive system is the communication protocol. Its correctness is central to enabling a user to develop applications with it confidently. Hence, several tests were created to verify this correctness. These are based on the tests also used in the reference implementation. The tests were all executed in Virtual Machines connected to the same internal network. We will first start by looking at the tests that were developed and with which our implementation was tested. Afterwards, a summary of the tests that are missing or were left out will be given.

5.2.1 Tests and results

For the implemented tests, a `vsomeip` application, called the `daemon`, was developed to widen the test scope. Even in existing automotive system, this application is often used as the central `vsomeip` routing component, here called the Host application. It is an application that does not offer or request any service and is merely mediating the communication between the local applications and the remote applications. Most of the tests can use this to force applications

to either use the `HostRoutingManager` or the `RoutingManagerProxy` component. In the former case, no `daemon` is needed while the latter requires a `daemon`. With this approach, a single test can be used to verify the behavior of different implementation paths, depending on the system configuration. Further logical branches can be tested by changing the applications network configuration, such as the usage of different transport protocols.

5.2.1.1 Application tests

These simple tests merely check the correct behavior of implementation-specific components. No AUTOSAR SOME/IP requirement is tested. The startup and stop mechanisms of a `vsomeip` application are tested, and the setup and execution of a user-defined watchdog to be called at a fixed interval. As the Rust implementation is more limited in these start and stop mechanisms, not all reference implementation tests were used.

- Start and stopping an application

  Starting an application once and then stopping it is the standard operating procedure and should not cause any issue.

- Start application twice

  Trying to start an application twice is not allowed and should print an error message.

- Stop application twice

  Stopping the application twice should do nothing and not raise any error.

- Watchdog setup and execution

  The user can register a single watchdog handler, only. Subsequent registrations override previous ones. The handler should be executed periodically at an approximately fixed interval defined by the user at registration. Using 0 as the interval should deregister an existing handler.

5.2.1.2 Service visibility tests

These 2 simple tests test that external applications cannot see a service instance configured to only be visible internally by applications in the same device. The other tests that a service instance configured to be visible externally can be discovered internally and externally.

- Locally visible service instance

  Create an application offering the local-only service instance and verify that an external application in a different device cannot find it

- Externally visible service instance

  Create an application offering the service instance and verify that an external application in a different device can find it

5.2.1.3 Service offer tests

This group of tests verifies the offering of services and error recovery for different kinds of application states. Most of the tests start with an application offering a service instance and a client application exchanging messages with the service. Then, the reaction of the system when different kinds of errors occur is tested. Except for the last test, all tests are only executed locally in the same device.

- Rejecting a local offer of a service still alive

  Offering the same service instance twice should not be accepted and an error message should be printed. Starting the second service application should not interrupt the client message exchange with the initial service application.

- Accept local offer after an application crash

  This test requires the `daemon` to be used as the system cannot renegotiate a Host application. So the central routing component would be lost, making it impossible for new local applications to be aware of each other existence. The test starts by initializing an offering application and a client that will be exchanging messages with the service offerer. The offering application will be forcefully terminated through a `KILL` signal, and both the Host application (`daemon`) and the client will recognize this crash. Once another application begins offering the same service instance, it should be accepted, and both the original client and any subsequent clients should initiate communication with this newly started application.

- Accept local offer on unresponsive application

  This test requires the `daemon` to be used as only the Host application can recognize unresponsive services. After the initial service application has become unresponsive, emulated through a `STOP` signal, a second application is started offering the same service instance. The second offer will be marked as pending and the `daemon` will ping the initial service application and expect a response within a specific time-frame. If it does not arrive, the initial service application is rejected, and the second offer is accepted, at which point all subsequent service clients will communicate with this new offerer.

- Reject local offer on already pending offer

  Like the previous test, we use the `daemon`, making the initial service application unresponsive through `STOP` and starting the second application offering the service, which will be marked as pending. Before the ping times out, start a third application offering the same service. This third offer should be rejected as there is already a pending offer from the second application. Subsequent service clients should communicate with the second service application, once the ping times out.

- Reject remote offer of already existing service

  Similar to the previous rejection test, but now involving remote applications offering the same service. Only one application should offer a service instance in the whole network, if a local application

already offers that service instance, then a remote offer should be rejected. Likewise, if a remote service instance already exists, then a new local offer of the same service should be rejected.

Note that the rejection of a service instance offer has a different meaning depending on the network topology. When multiple applications offer the same service instance, and both are running in the same device, there will be a device-wide consensus over which is the "correct" service instance. The first application that registers itself as the offerer of the service instance in the Host application will be the only one allowed to offer it in that device. Subsequent applications trying to offer the same service instance will be rejected, as long as the original offering application retains normal behavior. When we move to the external communication space, the rejecting behavior is different. Consider the hypothetical scenario, where 3 different devices are connected to the same network, 2 of them are executing applications offering the same service instance while the third one contains an application that intends to request that service instance. There is no network-wide consensus over which service instance is the "correct" one in such a situation. The rejection of offers in this situation can only be observed in both service offering applications through error messages that the user can read. The client application will keep alternating between using one service application and the other, according to which of them sent the most recent Service Discovery `OFFER` message. This behavior can also be observed by the user, as the client application will print warning messages each time the location (IP address and port) where the remote service instance can be reached is changed. Otherwise, it should continue to function correctly.

5.2.1.4 Requests test

The requests test certifies the correct request-response communication using SOME/IP. In it, 2 different devices are used, connected to the same network. Each device will contain 3 `vsomeip` applications, each offering a different service both through TCP and UDP. Once each application has discovered all the others, it starts sending 10 requests to each of them. Upon receiving a request, an application immediately sends back a response to the client. Neither the requests sent or the responses received have any payload. The test is a success when all applications have received a total of 50 responses, 10 per each other application.

There is no need to use the `daemon` in this test, as 3 applications per device mean that 2 of those applications will be using the `RoutingManagerProxy` component. However, different port configurations in a device are used. The default configuration is the use of different ports for each service. However, this can be mixed, and ports can be shared between service instances, so another configuration uses the same ports for UDP communication and the other using the same ports for both UDP and TCP communication.

5.2.1.5 Event notification test

The event test is aimed at testing the event behavior of SOME/IP. It only tests the SOME/IP Events and not the SOME/IP Fields. This test will start 2 applications, where one is the service provider with the event and the other the client that will subscribe to the event. The client registers the

appropriate handlers to react to different state changes in the network. Initially, the client waits for the service to become available, then it subscribes to the event and waits for the subscription to be accepted. Once that happens, it sends a request to the service that will serve as a signal to start sending the event notifications. This request will have as payload the number of notifications the server should send. This number could be made dynamic by reading it from the command-line when executing the client, but that is of little value, so the client always sends 50 as the payload. As such, the service will send exactly 50 notifications with the event value without any interval between them. The test is a success when the client receives those 50 notifications, at which point it sends another request to the server as a shutdown procedure.

The test is executed on different system configurations. It can be executed either locally on a single device or remotely on 2 different devices connected to the same network. The remote version can use the `daemon` to test the same behavior but using the `RoutingManagerProxy` component. For the local version there is no need as one of the applications will already be using the Proxy component. Furthermore, there exist 2 versions of this test, apart from using different transport protocols for communication. The Fixed Payload version, where the service notifications all have the same size, and the client tests whether the value received is the same size. In the Dynamic Payload version, the size of the payload of the notification sent by the service increases with each notification. The first notification will have 1 byte, the second 2 bytes and so forth. In this version, the client does not check the exact payload size, but only whether the size of the payload received is bigger than the size of the previously received payload. With this version, the ordering of not only the messages received but also the execution of the message handlers registered by the client is tested.

5.2.1.6 Initial field notification test

The other kind of events in the SOME/IP AUTOSAR specification are the Fields. As the main difference between them is the possibility of sending the Field initial value immediately after the subscription has been accepted, that is what is tested here. Once again, 2 different devices connected to the same network are used, each with 3 applications offering different service instances with one event each. These 3 service provider applications will startup and initially set the value of their respective Field. Afterwards, 10 client applications are started in each node that will subscribe to all of the Fields, including those in the same device. Each client will exit once it receives a notification from all of the Fields it has subscribed to, at which point the test is complete.

There are 2 tweaks possible to this test that change the behavior of the applications. The first is using multiple events per eventgroup as opposed to only a single one. A notification for all the events of the eventgroup shall then be sent to the client. The other is the timing of the subscription. By default, the client applications will only attempt to subscribe to the eventgroups once their respective service instances have become available. However, it is also possible to try to subscribe before that. In such a case, the subscription attempt will be registered at the Host application, which will then subscribe to the eventgroup as soon as possible. This allows the subscription to be sent amid the initial Service Discovery phase of the remote services.

5.2.1.7 Field notification test

To test the actual notifications of a Field event, these tests were made. Like the previous one, it uses 2 devices with 3 service applications, each offering a field event. Then, each service application subscribes to the other service applications fields and registers a subscription status handler, which will be executed each time the subscription is either accepted or rejected. Afterwards, all of the applications trigger 10 notifications that should be sent to each other. The test is a success if a service received the expected number of notifications for each subscribed field and if the subscription status handler was executed 5 times, representing the number of subscriptions it has made to the other 5 services.

The exact number of notifications received is dependent on which transport protocols are used for the notifications. For local communication, this is irrelevant as it will only receive a single notification regardless. For external services, if the eventgroup is offered in both UDP and TCP, it will send duplicate notifications, one through UDP and one through TCP, effectively doubling the expected notifications.

There is another different test for these notifications that uses more than one event per eventgroup. In this test, there is only a single application in each of the 2 devices. One application will be the service provider, offering 2 eventgroups and 3 events where one of the events belongs to both eventgroups. The other application will be subscribing to both eventgroups and receiving the notifications. After subscribing, the client will initially receive the "Initial Data" from the 3 events and then continuously set the fields value, which will trigger a notification in the service application side. The client is expected to only receive 1 notification per event, except for the "Initial Data" notification where it will receive 2 notifications from the event that belong to both eventgroups.

5.2.1.8 Selective event notification test

Selective events are not part of the AUTOSAR SOME/IP specification. They are specific to the GENIVI SOME/IP implementation to match the Franca selective broadcasts behavior. Nonetheless, as they are part of the suite of features offered by this implementation, they were also implemented in this work and as such have dedicated tests. Their test matches the Field notifications test and uses 2 devices with 3 service applications each offering a selective event. Each application will register 4 subscription status handler, testing all the possible combinations between service instance identifiers. All of the status handlers are valid for the subscriptions it will do, and as such, all 4 should be executed on a change of status. Afterwards, all applications will subscribe to the eventgroups of the remainder applications and send 10 notifications to each of the subscribers individually. The test is successful once the correct number of notifications has been received by all applications and when all of the handlers have been called. The number of notifications received, like the previous test, is also dependent on the transport protocol used.

5.2.1.9 Test results

Offer tests	Reject local offer			OK
	Accept offer on crash			OK
	Accept offer on unresponsive			OK
	Reject offer on already pending			OK
	Reject remote offer			OK
Requests tests	Different ports		TCP	OK
			UDP	OK
	Partial same ports		TCP	OK
			UDP	OK
	Same ports		TCP	OK
			UDP	OK
Event notification	Local	without `daemon`	Fixed payload	TCP OK
				UDP OK
			Dynamic payload	TCP OK
				UDP OK
	Remote	Using `daemon`	Fixed payload	TCP OK
				UDP OK
			Dynamic payload	TCP OK
				UDP OK
		Without `daemon`	Fixed payload	TCP OK
				UDP OK
			Dynamic payload	TCP OK
				UDP OK
Application tests	Start and stop			OK
	Start twice			OK
	Stop twice			OK
	Watchdog setup			OK

Field notification	1 Eventgroup per Event		TCP	**OK**
			UDP	**OK**
			TCP+UDP	**OK**
	2 Eventgroups for 1 Event		TCP	**OK**
			UDP	**OK**
			TCP+UDP	**OK**
Visibility tests	Locally visible			**OK**
	Externally visible			**OK**
Initial field notification	1 Event per Eventgroup	Subscribe on start	TCP	**OK**
			UDP	**OK**
			TCP+UDP	**OK**
		Subscribe on availability	TCP	**OK**
			UDP	**OK**
			TCP+UDP	**OK**
	5 Events per Eventgroup	Subscribe on start	TCP	**OK**
			UDP	**OK**
			TCP+UDP	**OK**
		Subscribe on availability	TCP	**OK**
			UDP	**OK**
			TCP+UDP	**OK**
Selective event notification			TCP	**OK***
			UDP	**OK***
			TCP+UDP	**OK***

In the last test of the table, the result column has an asterisk in every row. The reason for this is that the test did indeed pass, however, not in every execution. This occasional failure led to the belief that a race condition existed somewhere in the selective events code, hence the seemingly arbitrary results of the test. After probing the code in question, a race condition was indeed found. A sequence diagram of how this is triggered is shown in Figure 5.3. The scenario in question is similar to the selective events test, where there are 2 devices in the network, each with 2 SOME/IP

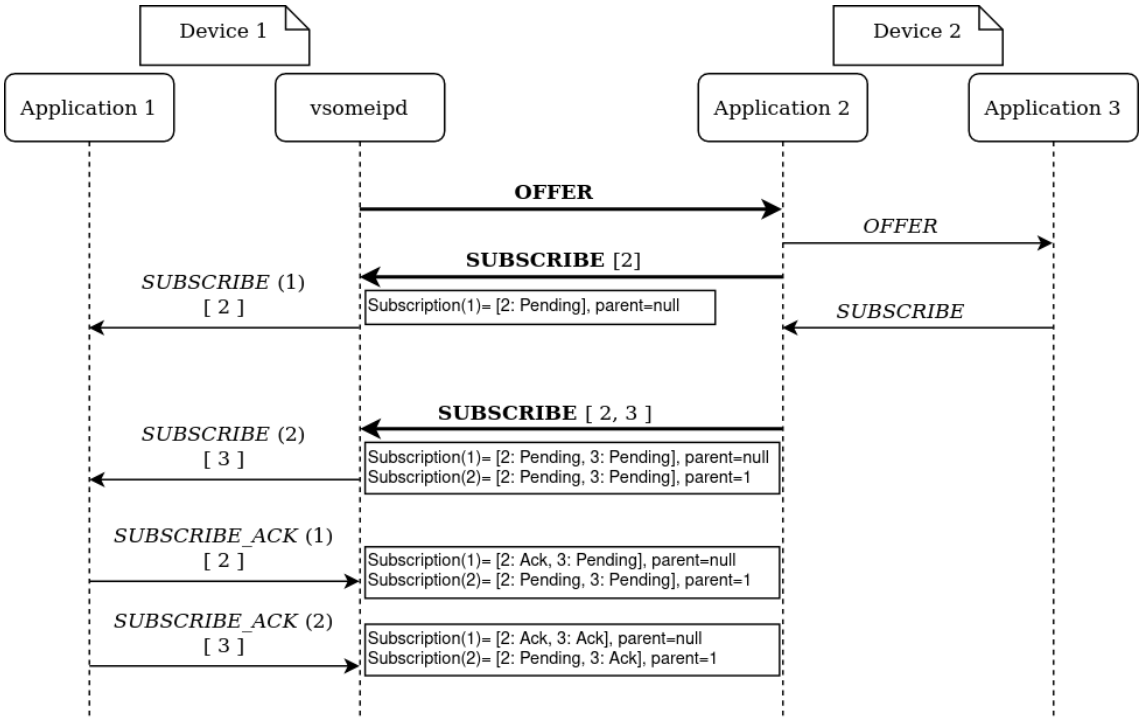


Figure 5.3: Sequence diagram of the detected race condition scenario. Bold messages are SOME/IP messages and italic messages are local messages. Inside the squares, `vsomeip` subscription status storage is shown.

applications running in them. Application 1 is offering a selective event, and `vsomeipd` is the Host application of device 1. This means that for every `SUBSCRIBE` message that `vsomeipd` receives from the network, it needs to notify Application 1 who will then decide whether to accept the subscription or not. The `vsomeipd` will only send back a `SUBSCRIBE_(N)ACK` message when it receives a response for this acceptance or rejection from Application 1. Now, assuming that both Application 2 and Application 3 want to subscribe to this remote selective event, the subscription happens in two steps. Since Application 2 is the Host application of Device 2, it will receive the `OFFER` message of the service instance that contains the event. It will then initially notify Application 3 of this new offer and send its own subscription to the event first. As this is a selective event, the information of which client has subscribed is sent in the `SUBSCRIBE` message, which, in the first one, is only Application 2, hence the [2]. For Application 3 to subscribe, it first needs to notify Application 2 of this, who will then add the client to the already subscribed clients of the remote event and then send another `SUBSCRIBE` message, now with both clients [2, 3].

Upon receiving the first `SUBSCRIBE` message, the `vsomeipd` will create a new subscription object, used to store remote subscriptions. Each of these objects is assigned a unique identifier, in this case, 1. As the first message only has client 2, it will only store client 2 in its remote clients, in a Pending state. It will then proceed to send the local `SUBSCRIBE` message to Application 1 with the new clients that subscribed and also the identifier of the remote subscription. The Pending state will only change to Acknowledged once Application 1 replies with a local

SUBSCRIBE_ACK message, as shown in the Figure. At that point, if all of the clients referenced in that subscription are Acknowledged, the SUBSCRIBE_ACK is propagated to Application 2. The second local SUBSCRIBE will only contain client 3, as it is the only new client from the newly received remote subscription. When this new remote subscription arrives, the original subscription will be updated with the new clients from the new subscription, and a reference to the original subscription is stored in the new subscription, as shown in the second memory state in the Figure. At the same time, the new subscription inherits the states of the clients of the original subscription.

This procedure poses no problem if the original subscription is processed by Application 1 before the arrival and processing of the second remote SUBSCRIBE message. However, if vsomeipd starts processing this second message before Application 1 processes the first one, then it will ultimately end in a state where no SUBSCRIBE_ACK is sent back to Application 2, due to the fact that no subscription would be fully acknowledged. Despite the subscription object 1 stored in memory, shown in the fourth memory state in the Figure, having all clients Acknowledged, it will not trigger the sending of the SUBSCRIBE_ACK message, since only the object referenced by the local SUBSCRIBE_ACK message is checked, and the subscription object 2 is not fully acknowledged and will never be. Having reached this state, subsequent SUBSCRIBE messages sent by Application 2 may or may not be answered. It depends on from which of the existing subscription objects, 1 or 2, the new message will inherit the states. Since both existing subscription objects are stored in a map that links the subscription identifier to the respective object, whether the new subscription will inherit the correct states depends on the order through which the map is iterated. Therefore, as maps are collections that give no guarantee to the user as to the ordering through which elements are traversed, we end up in an undetermined scenario.

This issue has already been reported back to the author of the library, [1] alongside a more detailed explanation linked with the code excerpts where this issue stems from. The author has acknowledged the error and a solution has already been found, so in the next release of vsomeip the issue should be resolved.

5.2.2 Missing tests

The tests mentioned above represent all the tests that were developed and with which our implementation was tested. These are mostly related to the correctness of the SOME/IP protocol itself. However, the reference implementation contains additional features mostly unrelated to SOME/IP that did not make it into this implementation. For each of those features there is a dedicated validation tests that was not implemented in this work.

5.2.2.1 Npdu feature test

The Npdu feature was added in GENIVI SOME/IP implementation as a means to trade lower network congestion for higher network latency. It serves as a caching buffer for outgoing messages so that they are not immediately sent. They are first cached and then at a fixed interval all the

[1] Issue #208 in the vsomeip repository

cached messages are sent. The test consists of a client sending messages to a service, and the service will be measuring if the client sends messages faster than it should be sending. As this feature was not implemented in this work, its corresponding test was also not developed.

5.2.2.2 Security test

The security features of GENIVI SOME/IP implementation allow the user to restrict which services can interact with each other. The user can specify that only a specific service instance is allowed to interact with another service instance. Every other client trying to interact with it will be rejected by the service. It is also possible to further restrict which clients are allowed to execute remote procedures or subscribe to events. This feature makes it much harder for an attacker to penetrate the system, but it does impose a need to have a more complete JSON configuration file with all these security specifications.

5.2.2.3 SOME/IP Transport Protocol test

The SOME/IP Transport Protocol is an additional SOME/IP protocol, like the Service Discovery, that is used to transport bigger payloads through UDP. If the payload does not fit in a single UDP packet then it will be fragmented into multiple messages on the sender side and then the receiver will reconstruct the message. The test is based on sending big messages to a remote service and that service ensuring that the reassembling of the message was done successfully. Although this protocol is part of the SOME/IP protocol it was not within the scope of this work and as such it was not implemented.

5.3 Implementation comparison

To further validate the correctness and feasibility of the Rust implementation, its performance should be at least similar to the reference implementation. Given the characteristics of Rust, this is possible, even if we would have done a one-to-one translation of C++ to Rust, which was not the case. As such, this chapter will focus on analyzing some of the runtime characteristics of both implementations. Although the implementations are different, their API is very similar, so a one-to-one translation of the test programs was made. Naturally, all of the tests were executed in the same environment using a high-end automotive System-on-Chip for In-vehicle Infotainment. To obtain the results of each test, 3 different executions were made. Between each execution, both the sender and the receiver applications were shutdown and then restarted. For each of the executions, we made a total of 12 repetitions. In each repetition, 10000 messages are sent from the client to the server, and the server will measure the time it took to receive and process all of those messages. Then, to rule out outliers, we removed the highest and the lowest value for each execution, ending up with a total of 30 values for each test parameter. All of the tests used TCP as the transport protocol to ensure all messages are received. An attempt at using UDP was made, but due to the nature of the protocol and the sheer number of messages being sent in a small time-frame, several

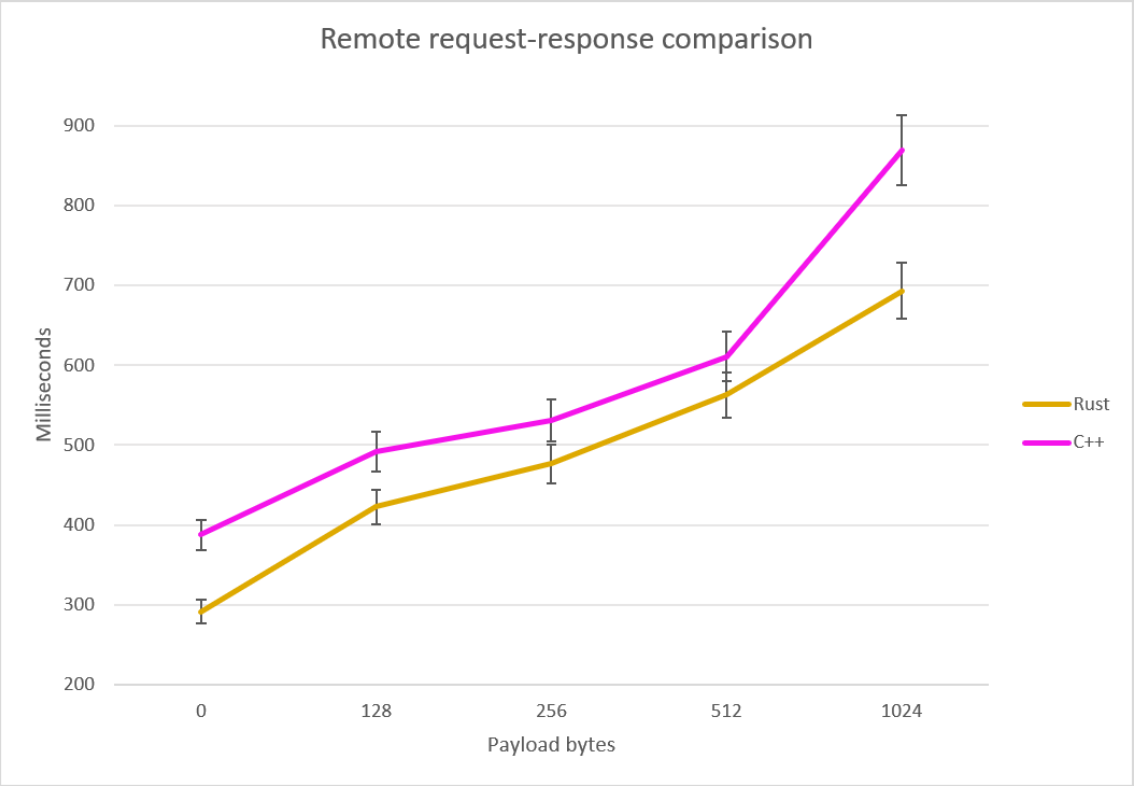
messages would be lost. This meant that the measurements would not be accurate and, as such, UDP measurement was not done. The full details of the system and the steps taken to build each of the applications are shown in Appendix A.

5.3.1 Request-response comparison

The first test made was related to the remote request-response communication paradigm in SOME/IP. A service application offers a certain service instance, which a second client application in a remote device will request. When the service becomes available, the client will first do a request to the `start()` method, which will signal the start time of the test. Immediately after, the client starts spamming the service with requests to the `spam()` method. In this method, the service only increments an atomic integer. All of the messages sent by the client are of `RequestNoReturn` type, so the service does not need to answer the request. As soon as the client finishes sending all of the spam messages, it ends with a request to the `end()` method. The service proceeds to read the duration that elapsed since the initial call to the `start()` method and prints out the results. This marks the end of a single repetition. The results of this test are shown in Figure 5.4 alongside the standard deviation.

An odd aspect of the Figure is that the table contains 3 rows of data, but the graph only contains 2 data sets. The data that is not shown is the "C++ (Sender)" data, as it would obscure the closer observation of the other 2 data sets. The values of this third data set were obtained by using a C++ client application and a C++ service application. The reason the results are so disparate to the other 2 data sets is the usage of the C++ client application. As previously mentioned in Section 4.3.1, the C++ SOME/IP implementation buffers outgoing messages, trading lower network congestion by higher network latency. The disparate results are precisely due to this feature. In the configuration file of this test, all of the `npdu-default-timings` were set to 0. This should cause the C++ client to buffer a message for 0 milliseconds, so it should be sent as soon as possible. However, as seen by the test results, that is not the case. Taking this into consideration, all of the tests involving remote communication used a Rust client to spam the messages. This reduces the amount of code that is actually compared between the implementations, but the results are fairer and easier to analyze. That is precisely the approach that was taken in the other 2 data sets, where we test the Rust sender with a Rust receiver and then a C++ receiver.

Between those 2 data sets, the results are more akin to one another, which, by itself, is a testament that it was a fair process. We see that the Rust implementation is consistently faster across all of the payload sizes, with the difference getting even bigger for the biggest payload size 1024 Bytes. This difference is likely due to 2 factors. The first is related to how each implementation handles the dispatching of its handlers, already detailed in Section 4.3.4. The application will register the `spam()` method as the handler for the spam messages the client sends. Since this method only increments an atomic integer variable, it will finish its execution very quickly. By barely spending any time executing the handler, the `wait_dispatch()` task, responsible for detecting a dispatching timeout, will likely not even have the chance to execute. When it does, the handler will already have finished its execution, so the main dispatcher already reacquired the



	0 Bytes	128 Bytes	256 Bytes	512 Bytes	1024 Bytes
C++	387 ±4,04%	492 ±3,18%	531 ±2,90%	611 ±3,47%	870 ±2,49%
Rust	291 ±2,11%	423 ±2,65%	476 ±2,70%	563 ±3,22%	693 ±3,91%
C++ (sender)	3816 ±3,06%	3878 ±2,22%	4045 ±1,54%	4048 ±2,02%	4111 ±1,46%

Figure 5.4: Comparison of the request-response communication between a Rust receiver and a C++ receiver

baton and `wait_dispatch()` will simply jump to the next loop iteration, without creating the new timeout object. On the other hand, the C++ implementation will always create a new timeout object regardless of how quickly the handler executes. This creation will undoubtedly have a cost that will be a constant value added to each message received.

The second differentiating aspect is also related to the sudden increase in the C++ timings from the 512 Bytes measurements to the 1024 Bytes. The service side of the SOME/IP implementations needs to deserialize the on-wire byte representation into a data type that the user can easily access. The key difference between the implementations is how this deserialization is made. For starters, in the C++ implementation, the deserialization is made through the aid of `deserializer` objects. At the startup of the program, a pool of these objects is created and stored behind a mutex. The deserialization is made at the beginning of the `deliver_message()` function, which will

```
bool deliver_message(const byte_t *data, length_t size, ...) {
  deserializer_mutex.lock();
  deserializer = deserializers.pop();
  deserializer_mutex.unlock();

  deserializer->set_data(data, size);
  its_message = deserializer->deserialize_message();

  deserializer_mutex.lock();
  deserializers.push(deserializer);
  deserializer_mutex.unlock();

  ...
}
```

Figure 5.5: C++ based pseudocode of the deserialization of a SOME/IP message prior to being delivered to the user's application

then trigger the execution of the message handlers. The initial lines of code of this function can be seen in Figure 5.5. As can be seen, the function will have to lock the mutex twice just to do the deserialization. In the scenario of the test, this does not pose much of a problem as the mutex will be uncontended. For each `Endpoint`, which is linked to a specific IP and port, only a single message is processed at a time. Since we are using only a single service instance linked to one IP and port pair, there will be no parallel processing of messages, so the mutex will remain uncontended throughout the test execution. For cases with multiple service instances and several messages arriving at the same time, this will not be the case, and the overhead of doing this will be much more significant. Regardless, in this test, the impact should be minor.

The main differentiating aspect of the deserialization is the need for a separate object to handle the deserialization. In the C++ implementation, it is handled by the `deserializer` data type that internally contains a vector used as a buffer for the deserialization. That is why the call to its `set_data()` method is required prior to the actual deserialization of the message. Inside this method, the object will copy all of the bytes received into its internal buffer through the call `buffer.assign(data, data + length)`. Only after this will the deserialization take place, where another copy of the message payload is made to the SOME/IP message object. In the Rust implementation, this initial copy of the buffer does not exist. In fact, there is not even a separate `deserializer` object. The deserialization is made through a static method that can be called anywhere in the code. The method directly takes as input the on-wire bytes received on the SOME/IP message and processes them without needing to first copy them. A copy to the separate SOME/IP message object is also required. With this approach, we are effectively cutting down 1 whole copy of the received message bytes and also the necessity to have any synchronization mechanisms. Since no other message from the same IP and port is processed at the same time, we are sure that the bytes received by the SOME/IP message will not change and thus can safely access them.

This explains the first sharp increase in timings that is felt from the 0 Bytes payload to 128

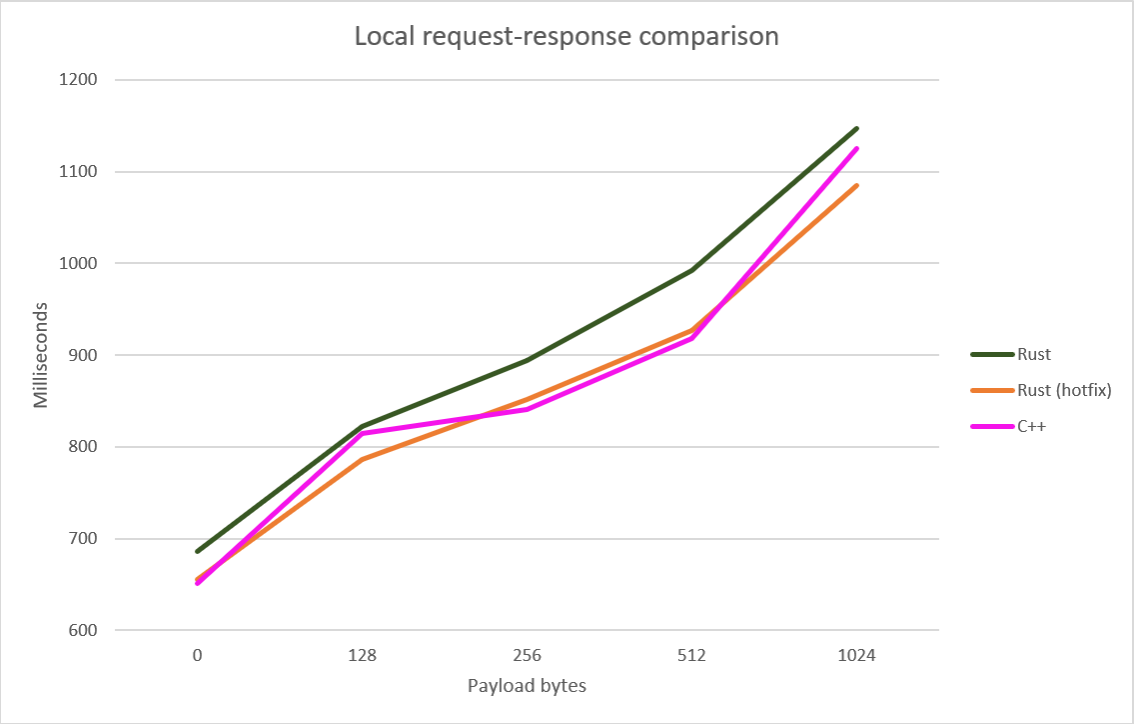
Bytes. Both languages will not allocate to the heap when the number of bytes in a collection is 0. Hence, when the SOME/IP message payload is copied to the message object, no allocation is required. However, as soon as at least 1 Byte is written, the collection will have to allocate, which has its impact in the application performance. Afterwards, from 128 Bytes to 512 Bytes, the increase in the timings is somewhat constant between both languages. The biggest spike is from the 512 Bytes to 1024 Bytes, where the cost of that additional copy in the C++ implementation is more significant. In contrast, the increase in timings in Rust remains similar from the 128 Bytes to the 1024 Bytes, so it is expected that for even higher payload sizes, the C++ implementation would suffer an even bigger increase in timings.

5.3.1.1 Local communication

By using the same test procedure as before, but using local communication instead of remote communication, we can test more components. Since the nPDU feature of the C++ implementation does not apply for local communication, no buffering of messages is done, and we can measure results for an all C++ network. This is what was done in this test, so the sender part of both implementations can be included in the testing. The results are shown in Figure 5.6..

In this test, we see that the Rust implementation and C++ are much closer in terms of performance. In fact, the base Rust implementation is consistently slower than the C++ implementation, as shown by the green color in the Figure. Taking into consideration the previous results of remote request-response, and knowing that the receiver applications use nearly the exact same code for local and remote communication, it stands to reason that the Rust slowdown is due to the sender application. However, exactly why this massive slowdown happens is unclear. One possible candidate was the way that the Rust implementation created the local messages. These need to start with a specific set of bytes and end with another specific set of bytes. In the Rust base implementation, the local message is first created and only afterwards framed with these bytes. Inserting the end frame is no problem as it does not cause any change in the message, it simply appends a few bytes at the end. The insertion of the start frame at the beginning of the message is more costly. Inserting anything at the beginning of an array causes the whole contents of the array to be shifted, resulting in a costly operation.

After obtaining the initial test results and noticing this performance disparity, a quick hotfix was made to the Rust implementation. It consisted of changing the creation of the `VSOMEIP_SEND` local messages to also include the framing, thus avoiding the shift. It was only applied to those messages as they are the ones used to transport the SOME/IP message between local applications. In this specific test, no other type of local message is sent. The timing results after this hotfix are what can be seen in the third row of data. This change did affect positively the timings, which are now on par with the C++ performance. Regardless, it still is a significant slowdown from the remote request-response communication results. To pinpoint exactly why that is the case would require a more thorough analysis of the Rust implementation, which could not be made. One aspect that would be worth studying that might be related to this slowdown is how demanding the



	0 Bytes	128 Bytes	256 Bytes	512 Bytes	1024 Bytes
C++	651 ±2,22%	815 ±3,38%	841 ±4,48%	918 ±4,89%	1125 ±2,66%
Rust	686 ±3,09%	822 ±2,27%	895 ±3,60%	993 ±2,49%	1147 ±2,67%
Rust (hotfix)	656 ±3,75%	786 ±2,56%	852 ±2,89%	927 ±4,05%	1085 ±2,22%

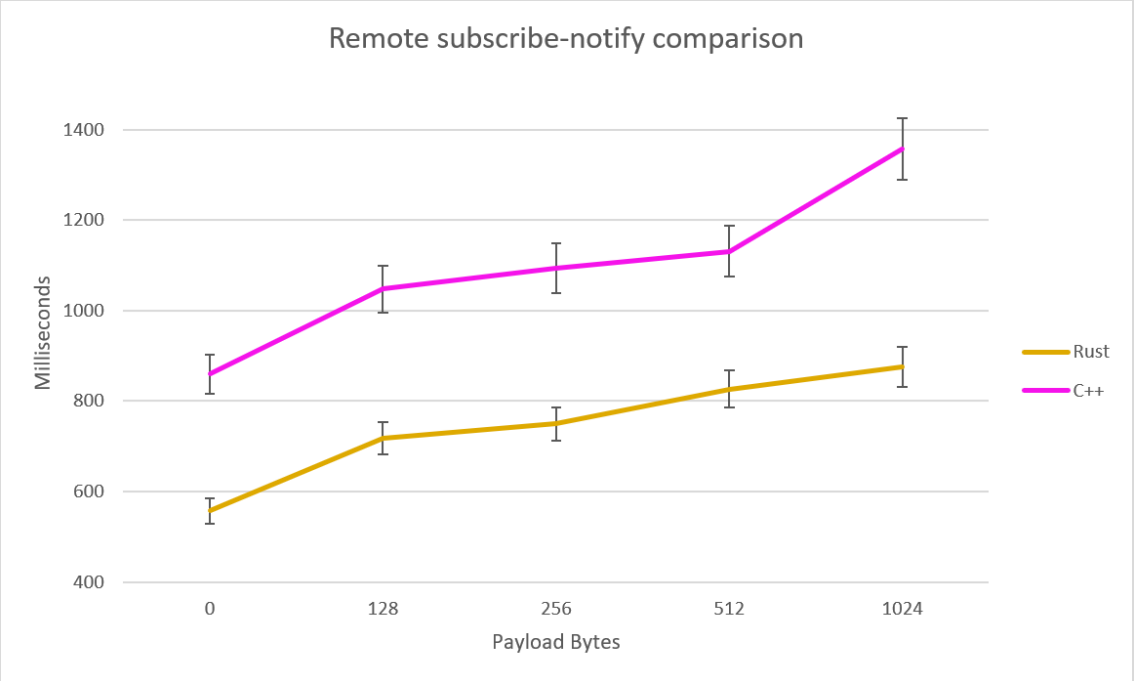
Figure 5.6: Comparison of the local request-response communication between the Rust sender-receiver and the C++ sender-receiver

Rust implementation is in terms of system resources. It might be the case that the Rust implementation is more demanding when compared to the C++, and so having 2 Rust applications executing in the same device hinders the performance of one another. If that is indeed the case, it needs to be fixed as that poses a major setback.

5.3.2 Subscribe-notify comparison

For the next test, the communication paradigm is different. Instead of request-response, subscribe-notify will be used. This also changes in which application the measurement is made. It will now be the service application that will be spamming the client with notifications. Similar to the previous tests, the service will be offering 3 events, one for starting the test measurements, another for the actual notification spamming and then the one to stop the measurements. The same 10000

notifications will be sent with an increasing number of Bytes in its payload. The results of the test are shown in Figure 5.7.



	0 Bytes	128 Bytes	256 Bytes	512 Bytes	1024 Bytes
C++	859	1047	1094	1131	1357
	±4,90%	±3,27%	±2,87%	±3,16%	±6,14%
Rust	557	718	749	827	875
	±2,36%	±2,12%	±3,48%	±2,59%	±4,90%

Figure 5.7: Comparison of the remote subscribe-notify communication between a Rust receiver and a C++ receiver

As can be observed, in this communication paradigm, the processing of notifications is nearly twice as slow as methods. Also, the Rust implementation is much faster at processing the notifications when compared to the reference implementation. Even with a 1024 Bytes payload, the Rust implementation is only slightly slower than C++ with 0 Bytes. One aspect that contributes to this is the different deserialization mechanisms which were previously explained. Another aspect that influences the results is the preference of the C++ implementation to copy a mutexed collection instead of holding the mutex for a while longer. Throughout both implementations, numerous objects have an internal state which stores some collection, for example, the `Event` object stores a collection of the eventgroups it belongs to. Access to these collections is protected by a mutex in both implementations. What differs is that most of the times, the C++ implementation prefers to lock the mutex, copy the whole collection, unlock the mutex and then iterate the newly created copy of the collection. In contrast, Rust implementation favors holding the mutex for a while longer and thus directly iterating the collection without having to copy it. Naturally, this

cannot be done everywhere as there are locations where, while iterating the collection, we also have to mutate it. In those cases, both implementations first copy the collection and then mutate it. While developing the Rust implementation, some comments were left in places where this copy was replaced by holding the mutex a while longer. In total, there are at least 43 places where a heap allocation is saved just by holding the mutex for a longer time. Most of these cases are in the Service Discovery logic and its impossible to measure their impact from a user application level. However, one of those cases is precisely in the processing of incoming notification messages where an allocation to the heap is saved. This certainly contributes to the speedup seen in the Rust implementation, but it should not be significant enough to the point of causing such a difference. What is likely causing such a major disparity, as it is the other main difference between implementations, is the C++ collection of event statistics, which was only added in version 3.1.20.1[2] It is unclear to the author exactly what these statistics are since they were added after the Rust implementation had been mostly finished. Nonetheless, in the newly added `insert_event_statistics()` function, regardless of whether statistics are activated or not in the configuration file, they keep being collected. This, as usual, involves a locking of a mutex, followed by several operations in the statistics collection. No doubt that these operations cause a major lag for the C++ implementation, despite them not being used for anything unless statistics are active in the configuration file. Unfortunately, the addition of this method, cast disbelief in the results obtained in this test as it is unclear exactly how much it affects the overall performance of the C++ implementation. In any case, it is made clear that in both implementations, the subscribe-notify communication paradigm is much slower than the usage of methods. Thus, we can at least draw the conclusion that for time-sensitive procedures, methods should be the preferred approach as opposed to notifications. This is something that seemed logical, so these results only confirmed existing suspicions.

5.3.3 Data types deserialization

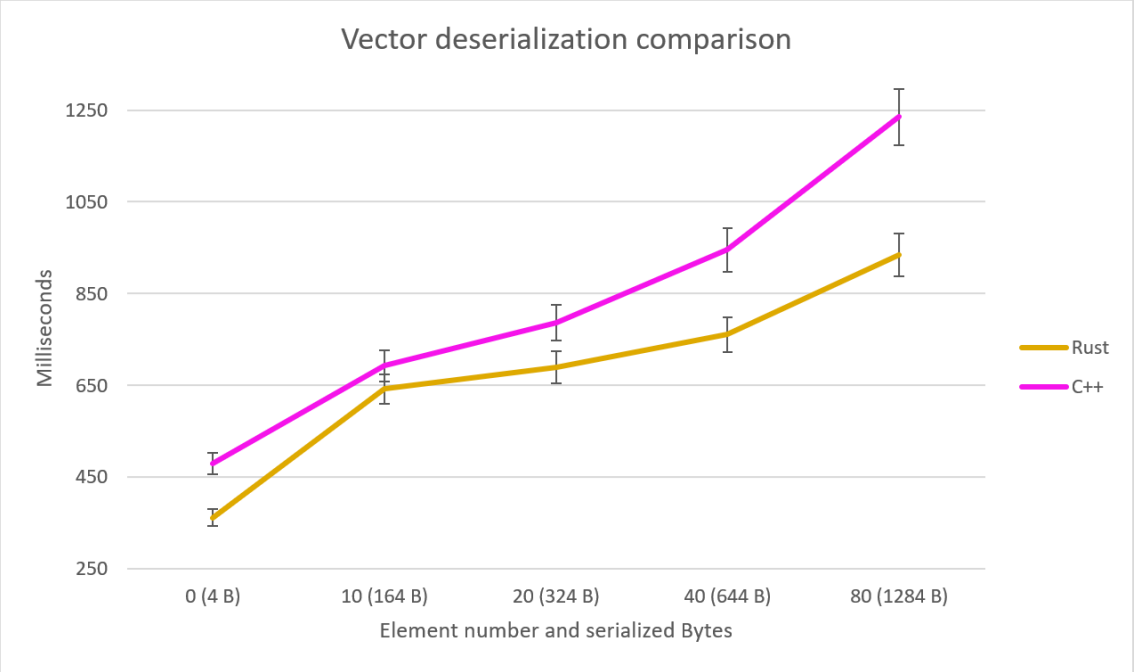
Having taken a look at the base SOME/IP protocol performance without using any remaining CommonAPI tools, some focus will now be given to the CommonAPI Runtime implementations. Specifically, what can be compared is the deserialization of the different data types. The tests follow the same procedure as the previous ones, 3 executions with 12 repetitions, in each repetition 10000 messages are sent. Only the request-response communication will be used. The messages are, once again, `RequestNoReturn`, and a Rust sender is always used. Hence, we already have the base timings for the SOME/IP protocol, as shown in Section 5.3.1 and can take them into consideration in the following tests.

5.3.3.1 Vector of structs deserialization

The first test uses a method that takes as input a vector of structs. An increasing number of elements is inserted in the vector that was chosen to be identical in their on-wire size, to the

[2] As seen in the `routing_manager_impl.cpp` file of that.commit

previous payload sizes explored. The declaration in the `.fidl` file of the method is shown in Figure 5.9 and the results in Figure 5.8.



	0 (4 Bytes)	10 (164 Bytes)	20 (324 Bytes)	40 (644 Bytes)	80 (1284 Bytes)
C++	480	692	787	945	1235
	±4,90%	±3,27%	±2,87%	±3,16%	±6,14%
Rust	362	642	690	761	934
	±2,36%	±2,12%	±3,48%	±2,59%	±4,90%

Figure 5.8: Comparison of the deserialization of a vector of structs between the Rust implementation and the C++

As can be seen, the results follow closely to what was seen previously in the remote request-response measurements in Section 5.3.1. For a vector with 0 elements, the timings are lower, due to the saved heap allocation. Then, with the increasing number of bytes, the disparity between implementations grows bigger. Up to the 20 elements in the vector, the difference in performance between the implementations remains similar to the reference remote request-response measurements. For the 40 element test, the difference grows, as we are now passing more than 100 additional Bytes when compared to the reference measurements. Also, as the number of Bytes increases, so does the amount of time the application spends executing the message handler and processing the Bytes of the message. This also means that there is a higher chance for the handler to be preempted by the kernel. Taking this into consideration, it would seem that both implementations are similar in their deserialization procedure. As previously mentioned, this is what was expected as the Rust CommonAPI (de)serialization procedures are roughly the same as the C++ ones. In the last test with 80 elements, the difference grows larger, however, so does the associated

```
struct Numbers {
  UInt32 u_32
  UInt64 u_64
  Float f_32
}
array NumbersVec of Numbers
method vec_serialization {
  in {
    NumbersVec arg
  }
}
```

Figure 5.9: Excerpt of the `.fidl` file section with the declaration of the method used to test vector deserialization

error in both implementations. Again, this is likely due to preemption from the kernel. When considering this, the difference seems to remain similar to the reference measurements.

5.3.3.2 Map of string to enumeration deserialization

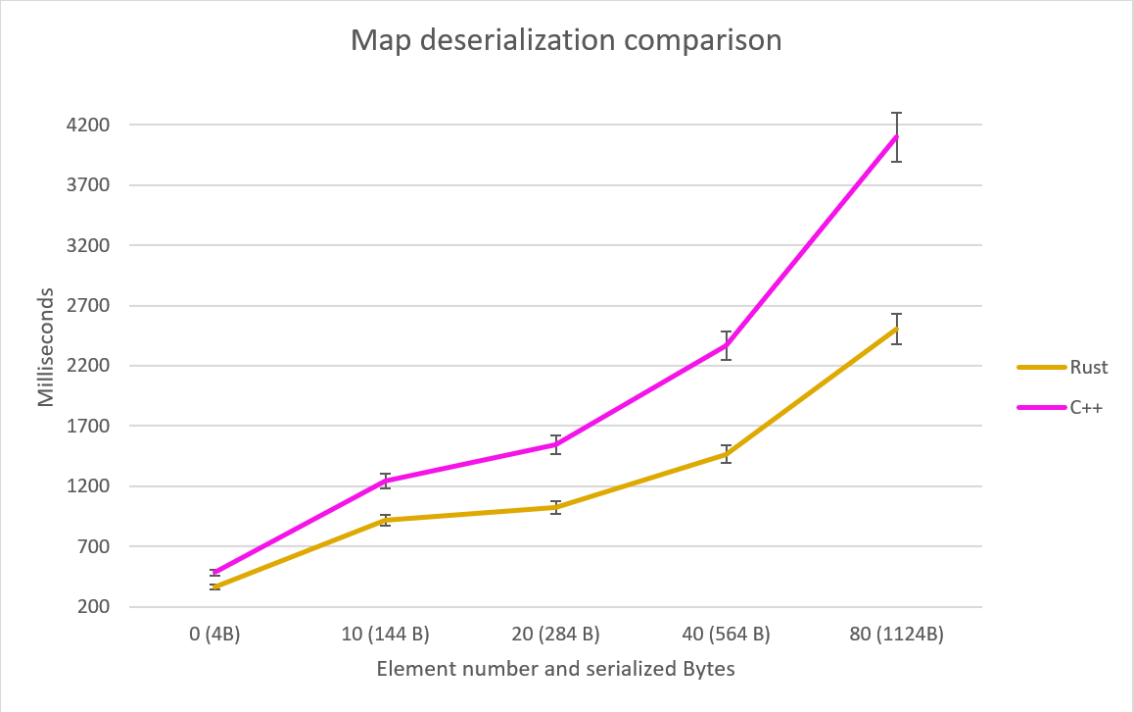
```
map StatusMap {
  String to Status
}

method map_serialization {
  in {
    NumbersVec arg
  }
}
```

Figure 5.10: Excerpt of the `.fidl` file section with the declaration of the method used to test map and string deserialization. The enumeration `Status` is omitted for simplicity sake.

The last test that was made now tests the deserialization procedure of a map. Similar to the previous test, a method was declared that takes a single input argument, as shown in Figure 5.10. The map will be associating a string to an enumeration. The enumeration declaration is not shown in the Figure to save space and because it is not very relevant to this test as it will simply be treated as a number when deserializing. All the strings that are inserted in the map have a total of 5 randomly generated alphabet letters. The default serialization arguments are used, so the string will be serialized as UTF-8 bytes, so each letter will be serialized into a single Byte. The results of the test are shown in Figure 5.11.

The results obtained are definitely astonishing and hard to believe. Several additional executions of the test were made to ensure that the result was not a mistake, but similar values were measured regardless. Once again, the 0 Bytes case is very similar between implementations and corresponds to the reference remote request-response results. However, as we increase the number of elements in the map, the disparity grows much quicker than what was expected. The 80 element



	0 (4 Bytes)	10 (144 Bytes)	20 (284 Bytes)	40 (564 Bytes)	80 (1124 Bytes)
C++	483 ±4,70%	1240 ±4,87%	1548 ±3,34%	2370 ±2,47%	4100 ±2,59%
Rust	364 ±2,84%	919 ±4,87%	1023 ±3,34%	1465 ±2,47%	2506 ±2,59%

Figure 5.11: Comparison of the deserialization of a map of strings to enumeration between the Rust implementation and the C++

test results of Rust implementation are nearly as fast as the C++ 40 element results. After double-checking the map and string deserialization procedures of both implementations, no distinction was found between them that could cause such a huge gap in performance as both use roughly the same procedure.

Since both implementations follow the same procedure and the timings of a request-response communication have already been measured and are nowhere near the values measured in this test. The other possible explanation for this disparity of performance must be in the programming languages themselves. The 3 types of data that are used in this test are the map, string and an enumeration. In both languages, an enumeration is simply stored as an integer, so its performance impact should be minimal. For a string, both store it using a UTF-8 encoding and the creation and insertion in a string are likely the same between languages as it is backed by an array which does not require any additional logic for creation or insertion. The last option is the map implementation. In the C++ CommonAPI implementation, they are represented as a `std::unordered_map` while in Rust as a `HashMap`. The map collection is much more complex than the vector that was analyzed before, an enumeration or a string. There are several algorithms that can be used to handle the insertion, each with their own advantages and disadvantages. As such, a conclusion that can be drawn is that the Rust and C++ implementations of a map collection are different. For the Rust implementation, as of this date, the algorithm used is a Rust port of Google's SwissTable algorithm.[3] According to the conference talk, linked in the Rust documentation, Google's C++ implementation of this algorithm yields 2 to 3 times better performance with significant memory reductions when compared to `std::unordered_map`. Thus, this is likely what is causing the big difference between the performance of the two implementations. It was an unintended improvement, but, nonetheless, it is there. In a sense, this reinforces the idea that when using the C++ standard library, only the `std::vector` collection should be used, as the others are not as optimized as they could be.

5.4 Summary

This chapter was dedicated to the validation of the SOME/IP implementation done in this work. For that, we demonstrated the implementation of the example system shown in Chapter 3. Through it, it was shown how the different developed CommonAPI components interact with each other and how interoperability with the reference C++ implementation is aided by them. Afterwards, a series of validation tests were analyzed, first covering the tests that were implemented in this work and then the ones that were not. Each of them with a detailed explanation of exactly what are they testing and why it is relevant. As a result of one of these tests, a race condition was detected that affected both the Rust implementation and the C++, which has been subsequently reported to the author of the library. In the last section of the chapter, we presented an objective comparison between both implementations through the means of stress testing them. It was shown that the Rust implementation performs at least as well as the reference implementation, with some

[3] As explained in the Rust documentation

scenarios where it even outperforms it. An explanation was made for each of these scenarios to clarify these differences in performances.

Chapter 6

Conclusion

Throughout this document, we have taken a look at the automotive software development world. It started as a relatively simple area with a low impact in the automotive industry. Throughout the years, this has changed, and it has now become a major area of a vehicle. To cope with this development, a new architecture of automotive software is now being used across the whole industry. By using a Service-Oriented Architecture, the developed software has a higher modularity, making the integration with existing software and also software that will be developed in the future very straightforward. One of the communication protocols that was created precisely for this dynamic nature and type of architecture was SOME/IP. This work focused on that protocol and how a Rust implementation of it can be used to develop robust automotive software.

This started with the analysis and exploration of the already existing approaches to developing software using SOME/IP. The reference for this work is the GENIVI CommonAPI, the standard open-source library used in the industry to develop applications using SOME/IP. Through it, an application can be abstracted not only from how the remaining applications in the system are developed, as a consequence of a Service-Oriented Architecture, but also from which communications protocol to use. The process of developing an application using CommonAPI begins by the specification of the services interfaces through Franca files. These files are then used as the standard that applications of the system need to follow to be able to communicate with one another. The CommonAPI provides a couple of code generators that read these Franca files and outputs the respective skeleton components that the user can then use to start developing its application. The resulting code is aided by the CommonAPI Runtime library, which provides an implementation of the Franca concepts for each supported communication protocol. The application will then use the CommonAPI Runtime abstraction layer so the user can develop the application without worrying about which communication protocol is being used. Beneath the CommonAPI Runtime layer, there are the actual communication protocols, with SOME/IP being the spotlight of this work.

The study of the results of this work followed a top-down approach, starting from the developed Rust code generator and ending with the Rust SOME/IP implementation. The resulting Rust code generator was a merge between the 2 CommonAPI code generators and was heavily based on the reference generators. In the Rust CommonAPI implementation, we explored how the various

abstraction layers provided by the CommonAPI were translated to Rust despite the less flexible typing system when compared to C++. Additionally, there was a simplification of several parts of the CommonAPI logic that make it much simpler to understand without incurring in any runtime cost for the Rust implementation. Lastly, the Rust SOME/IP implementation, as the core of this work, was analyzed in more detail. There, the architectural changes that occurred due to language differences were exposed and discussed. Furthermore, we delved into Rust paradigm regarding sharing of data by means of message passing, as opposed to the C++ approach of using mutual exclusion primitives.

Finally, the validation of the results of this work was made. It started by the development of an example system presented early in the document. This system showcased the whole CommonAPI workflow and how it can effectively be used to develop Rust code that can interoperate with the reference C++ implementation. Then, we showed the more extensive validation tests that were used to certify the developed Rust SOME/IP implementation does what is expected. Through this process, an error in both implementations was found which has been fixed in the Rust implementation and reported in the reference implementation. The validation ended with a more quantitative comparison of both implementations, where their performance was put to the test. We observed that the Rust implementation is at least as performant as the reference C++ implementation, even surpassing it under certain scenarios.

6.1 Summary of the results

Overall the results obtained in this work met all of the objectives set when planning it. During the planning phase, the original intention was not to develop a CommonAPI library that, like the reference implementation, was able to fully abstract over the underlying middleware. Nonetheless, that was able to be achieved, and all of the bedrock for any additional middleware is built. Regarding the SOME/IP implementation, originally the implementation of the Service Discovery protocol was also not planned. However, once we realized that a subscribe-notify communication paradigm was not possible without implementing the Service Discovery module, the original plan was updated. This happened fairly early in the development process, about 1 month after it began. So while it ultimately did not pose much of a problem, it still added to the total amount of work that had to be done.

One of the initial requirements, and perhaps the most crucial one, was the interoperability with the reference implementation. As undoubtedly the reference SOME/IP implementation is still used and will likely remain in use for a long time, it was of the utmost importance that the Rust implementation could smoothly integrate with existing systems. The other requirement was that the Rust implementation have similar runtime characteristics to the reference C++ implementation. From a performance point of view, this does seem to be the case, and it even surpasses the reference implementation under certain scenarios. Nonetheless, it might be the case that the Rust implementation is more resource-heavy when compared to the C++ implementation as seems to

be suggested by the results in Section 5.3.1.1 with the comparison of the local communication between the two implementations.

An unexpected, but very much welcome result of this work is the discovery of an otherwise elusive race condition in the SOME/IP implementation. With this discovery, we can at least say that we have given something back to the `vsomeip` community. As our work builds on top of the already existing projects from GENIVI, it stands to reason that we also help it as much as we can. Hence, this discovery is a very much welcomed result as it allowed us to contribute to the reference implementation, which is nowadays used throughout the automotive software industry.

6.2 Future work

The work developed in this project shows good promise and the author is of the firm belief that it is a step in the right direction for the automotive industry. Nevertheless, there are still some rough edges that need to be worked on. First and foremost, more extensive testing of the implemented CommonAPI libraries needs to be made, specially the code generators that, indirectly, also test the CommonAPI Runtime. As was explained, the main focus of this work was the SOME/IP implementation, so those 2 upper libraries were left in a minimum viable product state. While they certainly work for the most common scenarios, it is likely that there are some errors or mistakes that were made and not caught. Specifically, in the CommonAPI Runtime, the implementation of the asynchronous operations still needs to be finished. The architecture is already there, so it should be fairly simple, but it still needs to be done.

In the Rust SOME/IP implementation, the development status of the protocol itself is good and seems to work smoothly. However, there are certain `vsomeip` features, not related to the protocol that could be integrated into the Rust implementation to provide a more complete library. Namely, the possibility of stopping an application and then starting it once again, to effectively reboot the library. This will likely involve some non-trivial changes to the Rust code as it originally was not developed with this feature in mind. Then, the `vsomeip` nPDU feature that buffers the outgoing messages could also be integrated. Unlike the C++ implementation, though, this feature should be a compile-time option that can either be turned on or off. This would allow the user to decide whether to trade higher network latency by lower network congestion, like the reference implementation, or the opposite as is done with the Rust implementation. Having this as a compile option would mean that whichever option is chosen, the other one will have no effect on the runtime characteristics of the application.

Lastly, more testing and analysis of precisely the runtime characteristics of the SOME/IP implementation need to be made. This would first start with comparing how resource-heavy each implementation is, as the testing of the local communication suggests a significant difference might exist. Another type of testing that would be of interest is throughput testing, where an increasing number of requests are sent to a service application until its maximum capability to answer these requests is reached. The places that are causing this bottleneck can then be analyzed and further improved. This type of information gathering, followed by correcting mistakes or sub-optimal

solutions is reasonably lengthy and would require several trials and errors. Taking these aspects into consideration, additional future works could use the initial analysis made in this work as the stepping stone for a more profound analysis.

Appendix A

Setup procedure and details for Chapter 5 tests

All of the tests of Section 5.3 were made in a Renesas R-Car-H3 [1]. 2 Virtual Machines were flashed in it with TCP/IP communication between one another. Both VMs were using the same Linux version:

```
$> uname -a
    Linux ebcore-vm 4.19.72-eb-corbos-standard #1 SMP PREEMPT aarch64 GNU/Linux
```

A.1 C++ test binaries compilation

In order to use the reference `vsomeip` and CommonAPI libraries, they had to be recompiled from scratch. This included the `boost` library used by `vsomeip`. A compiler and linker of Elektrobit GmbH (EB) were used that is based on version 8.3.0 of `gcc` for ARM. This compiler is used both to compile the C++ programs and as a linker for the Rust compiler. The following steps were taken to compile `boost` version 1.74.0, fetched from the official website [2]:

1. Bootstrap the `boost` compiler:

   ```
   $> sh bootstrap.sh
   ```

2. Change environment to use target platform libraries (part of the provided EB compiler toolchain)

   ```
   $> source environment-setup-aarch64-poky-linux
   ```

3. Update the `project-config.jam` file in the root `boost` directory

   Replace the line

   ```
   using gcc ;
   ```

[1] Full R-Car H3 details and specifications
[2] Taken from here

With

```
using gcc : arm : aarch64-poky-linux-g++ -mcpu=cortex-a57.cortex-a53+crc --sysroot
    =/opt/corbos-linux/2.14.0/sysroots/aarch64-poky-linux
```

So that compiling `boost` to the toolset `gcc-arm` will use the specified compiler

4. In the root directory execute

```
$> ./b2 toolset=gcc-arm variant=release
```

5. The resulting shared object files will be put in the `stage/lib` directory

The `vsomeip` version used was 3.1.20.3, the latest version as of this writing and fetched from the official repository [3]. Inside the root directory of the `vsomeip` library:

1. Change environment to use target platform libraries (part of the provided EB compiler toolchain)

```
$> source environment-setup-aarch64-poky-linux
```

2. Create directory to hold compilation files

```
$> mkdir arm_build
```

3. Change current directory to the newly created

```
$> cd arm_build
```

4. Generate the build system

```
$> cmake -DCMAKE_BUILD_TYPE=Release ..
```

5. Compile `vsomeip`

```
$> make
```

6. The resulting shared object files will be in the same directory, `arm_build`.

For the CommonAPI Runtime, version 3.2.0 of `capicxx-core-runtime` was used, fetched from the official repository [4]. Inside the root directory of the `capicxx-core-runtime` library:

1. Change environment to use target platform libraries (part of the provided EB compiler toolchain)

```
$> source environment-setup-aarch64-poky-linux
```

2. Create directory to hold compilation files

```
$> mkdir arm_build
```

[3] Taken from here
[4] Taken from here

3. Change current directory to the newly created

```
$> cd arm_build
```

4. Generate the build system

```
$> cmake -DCMAKE_BUILD_TYPE=Release ..
```

5. Compile `capicxx-core-runtime`

```
$> make
```

6. The resulting shared object files will be in the same directory, `arm_build`.

For the CommonAPI SOME/IP Runtime, version 3.2.0 of `capicxx-someip-runtime` was used, fetched from the official repository [5]. Inside the root directory of the `capicxx-someip-runtime` library:

1. Change environment to use target platform libraries (part of the provided EB compiler toolchain)

```
$> source environment-setup-aarch64-poky-linux
```

2. Create directory to hold compilation files

```
$> mkdir arm_build
```

3. Change current directory to the newly created

```
$> cd arm_build
```

4. Generate the build system

```
$> cmake -DCMAKE_BUILD_TYPE=Release ..
```

   CMake options pointing to the `arm_build` folders of the previously compiled `vsomeip` and `capicxx-core-runtime` also need to be provided. This is not shown in the command above for simplicity sake.

5. Compile `capicxx-someip-runtime`

```
$> make
```

6. The resulting shared object files will be in the same directory, `arm_build`.

With these 4 libraries, we now have everything we need to start compiling the test programs. For the tests shown in Section 5.3.1 and 5.3.2 the CommonAPI Runtime libraries are not required. Whereas for the tests in Section 5.3.3 all of the compiled libraries are required. For the former type of tests, the following command was used to compile them:

[5] Taken from here

	capicxx-core-tools	capicxx-someip-tools
Version	3.2.0	3.2.0
Build ID	20200114	20200114
Architecture	64bit	64bit
Franca version	0.13.1	0.13.1
CommonAPI version	3.2.0	

Table A.1: Details of the CommonAPI code generators used

```
$> $CXX -O3 <name> -o arm_<name> -lvsomeip3
```

We use `$CXX` here instead of writing the full compiler options as shown in the 3rd step of compiling the `boost` library. This command assumes that all of the previously compiled libraries can be found in the system path. An additional compiler flag, `-lpthread` was required for some of the tests.

For the latter tests that use the CommonAPI Runtime, code generation had to be used. For that, the following versions of the code generators were used:

To generate the code, the files in B.4 and B.5 were used. Then, to compile them, the CMake file seen in B.6 was used with the following commands:

1. Change environment to use target platform libraries (part of the provided EB compiler toolchain)

   ```
   $> source environment-setup-aarch64-poky-linux
   ```

2. Create directory to hold compilation files

   ```
   $> mkdir arm_build
   ```

3. Change current directory to the newly created

   ```
   $> cd arm_build
   ```

4. Generate the build system

   ```
   $> cmake -DCMAKE_BUILD_TYPE=Release ..
   ```

5. Compile the test binaries

   ```
   $> make
   ```

6. The resulting shared object files will be in the same directory, `arm_build`.

A.2 Rust test binaries compilation

For the compilation of the Rust binaries, version 1.51.0-nightly (7a9b552cb) was used. To do the compilation, the following steps were taken:

1. Install the toolchain of the R-Car H3 platform

```
$> rustup target install aarch64-unknown-linux-gnu
```

2. Create a `config` file in the `/.cargo` directory with the following lines

```
[target.aarch64-unknown-linux-gnu]
linker="aarch64-poky-linux-gcc"
rustflags = [
"-C", "link-arg=-mcpu=cortex-a57.cortex-a53+crc",
"-C", "link-arg=--sysroot=/opt/corbos-linux/2.14.0/sysroots/aarch64-poky-linux"
]
```

   Used to tell the Rust compiler to use the linker provided by EB when compiling for the `aarch64-unknown-linux-gnu` target platform.

3. Change environment to use target platform libraries (part of the provided EB compiler toolchain)

```
$> source environment-setup-aarch64-poky-linux
```

4. In the root of the Rust project run:

```
$> cargo build --release --all --target=aarch64-unknown-linux-gnu
```

5. The resulting binary files will be in the `target/aarch64-unknown-linux-gnu/release` directory.

This process applies to all of the Rust binaries used for the tests. All of them used the following release profile, specified in their respective `Cargo.toml` files:

```
[profile.release]
lto = "true"
panic = "abort"
codegen-units = 1
```

Each of the developed Rust libraries use additional packages, a full list of the packages used by each of the developed Rust libraries and their respective version is shown below:

- Rust `vsomeip` library
  - `arc-swap` v1.2.0
  - `async-trait` v0.1.42
  - `crossbeam-channel` v0.5.0
  - `crossbeam-utils` v0.8.1

  - `env_logger` v0.8.2
  - `fastrand` v1.4.0
  - `fs2` v0.4.3
  - `log` v0.4.13
  - `parking_lot` v0.11.1
  - `serde_json` v1.0.61
  - `socket2` v0.3.19
  - `tokio` 0.3.6

- Rust CommonAPI library (`capi_runtime`)
  - `arc-swap` v1.2.0
  - `crossbeam-utils` v0.8.1
  - `ctor` v0.1.18
  - `env_logger` v0.8.2
  - `lazy_static` v1.4.0
  - `log` v0.4.13
  - `num-derive` v0.3.3
  - `num-traits` v0.2.14
  - `parking_lot` v0.11.1
  - `vsomeip` (Rust SOME/IP implementation)

For all the release builds, the feature flag `max_level_off` was passed to the `log` package, to disable all logs at compile-time.

Appendix B

Files used throughout the work

```
package elektrobit.rust

interface InfotainmentHub {
    version { major 1 minor 0 }

    /* Receives a new frame to render on the screen */
    method new_frame {
        in {
            AutomotiveTypes.Frame frame
        }
    }

    broadcast shutdown {
        out {
        }
    }
}

interface Seats {
    version { major 1 minor 0 }

    /* Has all the information regarding the seats, including seatbelts */
    attribute AutomotiveTypes.SeatInformation seats readonly

    /* Updates heating information of a seat */
    method set_heating {
        in {
            String seat
            AutomotiveTypes.SeatHeating heating
        }
    }
}

interface Camera {
    version { major 1 minor 0 }

    /* (De)Activates the camera, triggering it to call InfotainmentHub::new_frame()
        repeatedly */
    method activate {
        in {
```

```
            Boolean to_on
        }
    }
}

interface Wheels {
    version { major 1 minor 0 }
    attribute AutomotiveTypes.AllWheels status readonly
    attribute Float speed readonly

    attribute Boolean sport_mode
}

typeCollection AutomotiveTypes {
    version { major 1 minor 0 }

    array Frame of Pixel
    array AllWheels of WheelStatus

    struct Pixel {
        UInt8 r
        UInt8 g
        UInt8 b
        Float a
    }

    enumeration SeatHeating {
        OFF
        BOTTOM
        BACK
        BOTH
    }

    map SeatInformation {
        String to Seat
    }

    struct Seat {
        Boolean seatbelt
        SeatHeating heating
    }

    union WheelStatus {
        Double pressure
        String damage
    }
}
```

Listing B.1: CompleteCar example `.fidl` file

```
import "platform:/plugin/org.genivi.commonapi.someip/deployment/CommonAPI-
    SOMEIP_deployment_spec.fdepl"
import "CompleteCar.fidl"

define org.genivi.commonapi.someip.deployment for interface elektrobit.rust.InfotainmentHub {
    SomeIpServiceID = 4660
```

```
    method new_frame {
        SomeIpMethodID = 1000
        SomeIpReliable = false
    }

    broadcast shutdown {
        SomeIpEventID = 59999
        SomeIpReliable = false
        SomeIpEventGroups = { 6999 }
    }
}

define org.genivi.commonapi.someip.deployment for interface elektrobit.rust.Seats {
    SomeIpServiceID = 4665

    attribute seats {
        SomeIpGetterID = 1005
        SomeIpNotifierID = 60000
        SomeIpEventGroups = { 7000 }
    }

    method set_heating {
        SomeIpMethodID = 1006
        SomeIpReliable = true
    }
}

define org.genivi.commonapi.someip.deployment for interface elektrobit.rust.Camera {
    SomeIpServiceID = 4670

    method activate {
        SomeIpMethodID = 1010
        SomeIpReliable = true
    }
}

define org.genivi.commonapi.someip.deployment for interface elektrobit.rust.Wheels {
    SomeIpServiceID = 4675

    attribute status {
        SomeIpGetterID = 1015
        SomeIpNotifierID = 60005
        SomeIpEventGroups = { 7005 }
    }

    attribute speed {
        SomeIpGetterID = 1016
        SomeIpNotifierID = 60006
        SomeIpEventGroups = { 7006 }
    }

    attribute sport_mode {
        SomeIpGetterID = 1017
        SomeIpSetterID = 1018
        SomeIpNotifierID = 60007
        SomeIpEventGroups = { 7007 }
    }
```

```
}

define org.genivi.commonapi.someip.deployment for typeCollection elektrobit.rust.
    AutomotiveTypes {
    array AllWheels {
        SomeIpArrayLengthWidth = 0
        SomeIpArrayMaxLength = 2
        SomeIpArrayMinLength = 2
    }
}

define org.genivi.commonapi.someip.deployment for provider Service {
    instance elektrobit.rust.InfotainmentHub {
        InstanceId = "hub"
        SomeIpInstanceID = 22136
    }

    instance elektrobit.rust.Seats {
        InstanceId = "seats"
        SomeIpInstanceID = 22137
    }

    instance elektrobit.rust.Camera {
        InstanceId = "camera"
        SomeIpInstanceID = 22138
    }

    instance elektrobit.rust.Wheels {
        InstanceId = "front_wheels"
        SomeIpInstanceID = 22139
    }

    instance elektrobit.rust.Wheels {
        InstanceId = "back_wheels"
        SomeIpInstanceID = 22140
    }
}
```

Listing B.2: CompleteCar example `.fdepl` file

```
[package]
name = "CompleteCar"
version = "0.1.0"
authors = ["joal272320"]
edition = "2018"

[features]
default = [ "someip" ]

someip = ["vsomeip", "capi_runtime/someip", "capi_macros/someip"]

[lib]
name = "base"
path = "src/lib/mod.rs"

[[bin]]
name = "hub"
```

```
path = "src/hub/main.rs"

[[bin]]
name = "wheels"
path = "src/wheels/main.rs"

[[bin]]
name = "camera"
path = "src/camera/main.rs"

[dependencies]
capi_runtime = {path = "../capi_runtime"}
capi_macros = {path = "../capi_runtime/capi_macros" }
vsomeip = {optional = true, path = "../vsomeip"}

ctor = "^0.1"
num-traits = "*"
num-derive = "*"
```

Listing B.3: CompleteCar example generated `Cargo.toml` file

```
package elektrobit.rust

interface StressTest {
    version { major 1 minor 0 }

    method start_test fireAndForget {
        in {
            UInt64 payload_size
        }
    }

    method end_test fireAndForget {
        in {}
    }

    method map_serialization fireAndForget {
        in {
            StressTypes.StatusMap arg
        }
    }

    method vec_serialization fireAndForget {
        in {
            StressTypes.NumbersVec arg
        }
    }
}

typeCollection StressTypes {
    version { major 1 minor 0 }

    enumeration Status {
        ZERO = 0
        ONE = 1
        TWO = 2
        THREE = 3
```

```
        FOUR = 4
    }

    struct Numbers {
        UInt32 u_32
        UInt64 u_64
        Float f_32
    }

    array NumbersVec of Numbers

    map StatusMap {
        String to Status
    }
}
```

Listing B.4: `.fidl` file used for the serialization tests

```
import "platform:/plugin/org.genivi.commonapi.someip/deployment/CommonAPI-
    SOMEIP_deployment_spec.fdepl"
import "StressTest.fidl"

define org.genivi.commonapi.someip.deployment for interface elektrobit.rust.StressTest {
    SomeIpServiceID = 4660

    method map_serialization {
        SomeIpMethodID = 1000
        SomeIpReliable = true
    }

    method vec_serialization {
        SomeIpMethodID = 1001
        SomeIpReliable = true
    }

    method start_test {
        SomeIpMethodID = 1002
        SomeIpReliable = true
    }

    method end_test {
        SomeIpMethodID = 1003
        SomeIpReliable = true
    }
}

define org.genivi.commonapi.someip.deployment for provider as Service {
    instance elektrobit.rust.StressTest {
        InstanceId = "Test"
        SomeIpInstanceID = 22136
    }
}
```

Listing B.5: `.fdepl` file used for the serialization tests

```
cmake_minimum_required(VERSION 2.8)
```

```
set(CMAKE_VERBOSE_MAKEFILE on)

OPTION(USE_FILE "Set to OFF to disable file logging" OFF )
message(STATUS "USE_FILE is set to value: ${USE_FILE}")

OPTION(USE_CONSOLE "Set to OFF to disable console logging" OFF )
message(STATUS "USE_CONSOLE is set to value: ${USE_CONSOLE}")

IF(USE_FILE)
  add_definitions(-DUSE_FILE)
ENDIF(USE_FILE)
IF(USE_CONSOLE)
  add_definitions(-DUSE_CONSOLE)
ENDIF(USE_CONSOLE)

SET(MAX_LOG_LEVEL "ERROR" CACHE STRING "maximum log level")
message(STATUS "MAX_LOG_LEVEL is set to value: ${MAX_LOG_LEVEL}")
add_definitions(-DCOMMONAPI_LOGLEVEL=COMMONAPI_LOGLEVEL_${MAX_LOG_LEVEL})

if (MSVC)
# Visual C++ is not always sure whether he is really C++
set(CMAKE_C_FLAGS "${CMAKE_C_FLAGS} -D_CRT_SECURE_NO_WARNINGS /EHsc /wd\\\"4503\\\"")
set(CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} -D_CRT_SECURE_NO_WARNINGS /wd\\\"4503\\\"")
else()
set(CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} -Wall -O3 -pthread -std=c++0x -
    D_GLIBCXX_USE_NANOSLEEP")
endif()

message(STATUS "Compiler options: ${CMAKE_CXX_FLAGS}")

if(NOT CMAKE_BUILD_TYPE)
   set(CMAKE_BUILD_TYPE "Release" CACHE STRING
       "Choose the type of build, options are: Debug Release." FORCE)
endif(NOT CMAKE_BUILD_TYPE)
message(STATUS "Build type: ${CMAKE_BUILD_TYPE}")

OPTION(USE_INSTALLED_COMMONAPI "Set to OFF to use the local (build tree) version of CommonAPI
    " ON)
message(STATUS "USE_INSTALLED_COMMONAPI is set to value: ${USE_INSTALLED_COMMONAPI}")

if ("${USE_INSTALLED_COMMONAPI}" STREQUAL "ON")
    FIND_PACKAGE(CommonAPI 3.2 REQUIRED CONFIG NO_CMAKE_PACKAGE_REGISTRY)
else()
    FIND_PACKAGE(CommonAPI 3.2 REQUIRED CONFIG NO_SYSTEM_ENVIRONMENT_PATH
        NO_CMAKE_SYSTEM_PATH)
endif()

message(STATUS "CommonAPI_CONSIDERED_CONFIGS: ${CommonAPI_CONSIDERED_CONFIGS}")
message(STATUS "COMMONAPI_INCLUDE_DIRS: ${COMMONAPI_INCLUDE_DIRS}")

# CommonAPI
include(FindPkgConfig)

# SOME/IP
find_package (CommonAPI-SomeIP 3.2 REQUIRED)
find_package (vsomeip3 3 REQUIRED)
```

```
# Source files
set(SRC_PATH src)
set(SRC_GEN_PATH src-gen/v1/elektrobit/rust)

# Service Sources
FILE(GLOB STRESS_PROXY_GEN_SRCS ${SRC_GEN_PATH}/StressTest*Proxy.cpp ${SRC_GEN_PATH}/*
    Deployment.cpp)
FILE(GLOB STRESS_STUB_GEN_SRCS ${SRC_GEN_PATH}/StressTest*Stub*.cpp ${SRC_GEN_PATH}/*
    Deployment.cpp)

# Boost
find_package( Boost 1.74 COMPONENTS system thread log REQUIRED )
include_directories( ${Boost_INCLUDE_DIR} )

# DBus library
FILE(GLOB PRJ_DBUS_LIB_SRCS ${PRJ_SRC_GEN_COMMONAPI_DBUS_PATH}/*cpp ${PRJ_TYPES_GEN_SRCS})

# SOME/IP library
FILE(GLOB PRJ_SOMEIP_LIB_SRCS ${PRJ_SRC_GEN_COMMONAPI_SOMEIP_PATH}/*cpp ${PRJ_TYPES_GEN_SRCS
    })

# Paths
OPTION(USE_INSTALLED_DBUS "Set to OFF to use the local (patched) version of dbus" ON)
message(STATUS "USE_INSTALLED_DBUS is set to value: ${USE_INSTALLED_DBUS}")

include_directories(
    src-gen
    ${COMMONAPI_INCLUDE_DIRS}
    ${COMMONAPI_SOMEIP_INCLUDE_DIRS}
    ${VSOMEIP3_INCLUDE_DIRS}
)

link_directories(
    ${COMMONAPI_LIBDIR}
    ${COMMONAPI_SOMEIP_CMAKE_DIR}/build
    ${Boost_LIBRARY_DIR}
)

set(LINK_LIBRARIES CommonAPI CommonAPI-SomeIP vsomeip3)

# Build StressTest Service
add_executable(service ${SRC_PATH}/service.cpp ${SRC_PATH}/SpammerService.cpp ${
    STRESS_STUB_GEN_SRCS})
target_link_libraries(service ${LINK_LIBRARIES})
```

Listing B.6: CMake file used to generate the build system of the serialization tests

References

[1] Gunnar Andersson. Commonapi overview. https://at.projects.genivi.org/wiki/display/DIRO/CommonAPI+overview, 2018. [Online; accessed 21-June-2020].

[2] AUTOSAR. SOME/IP Service Discovery Protocol Specification. https://www.autosar.org/fileadmin/user_upload/standards/foundation/19-11/AUTOSAR_PRS_SOMEIPServiceDiscoveryProtocol.pdf, year = 2019, month = November, note = "[Online; accessed 21-June-2020]".

[3] AUTOSAR. Explanation of Adaptive Platform Design. pages 1–84, November 2019. [Online; accessed 19-June-2020].

[4] AUTOSAR. Explanation of ara::com API. https://www.autosar.org/fileadmin/user{_}upload/standards/adaptive/19-11/AUTOSAR{_}EXP{_}ARAComAPI.pdf, 2019. [Online; accessed 22-June-2020].

[5] AUTOSAR. Integration of Franca IDL Software Component Descriptions. https://www.autosar.org/fileadmin/user{_}upload/standards/classic/19-11/AUTOSAR{_}TR{_}FrancaIntegration.pdf, November 2019. [Online; accessed 22-June-2020].

[6] AUTOSAR. Layered Software Architecture. pages 1–166, November 2019. [Online; accessed 16-June-2020].

[7] AUTOSAR. SOME/IP Protocol Specification. https://www.autosar.org/fileadmin/user{_}upload/standards/foundation/19-11/AUTOSAR{_}PRS{_}SOMEIPProtocol.pdf, November 2019. [Online; accessed 21-June-2020].

[8] AUTOSAR. Specification of Communication Management. https://www.autosar.org/fileadmin/user{_}upload/standards/adaptive/19-11/AUTOSAR{_}SWS{_}CommunicationManagement.pdf, 2019. [Online; accessed 22-June-2020].

[9] AUTOSAR. Specification of RTE Software. pages 1–1313, November 2019. [Online; accessed 16-June-2020].

[10] AUTOSAR. Specification of SOME / IP Transformer. https://www.autosar.org/fileadmin/user{_}upload/standards/classic/19-11/AUTOSAR{_}SWS{_}SOMEIPTransformer.pdf, November 2019. [Online; accessed 21-June-2020].

[11] Stefano Buliani. Rust runtime for aws lambda. https://aws.amazon.com/blogs/opensource/rust-runtime-for-aws-lambda/, November 2018. [Online; accessed 24-June-2020].

[12] Redox Developers. https://www.redox-os.org/, 2020. [Online; accessed 24-June-2020].

[13] Electronics Engineers and Three Park Avenue. Draft Standard for 9 Local and metropolitan area networks — Time-Sensitive Networking Profile for 11 Automotive In-Vehicle Ethernet 12 Communications. pages 1–67, 2020.

[14] Jeremy Fitzhardinge. Bringing rust home to meet the parents. https://www.youtube.com/watch?v=kylqq8pEgRs, September 2019. [Online; accessed 24-June-2020].

[15] Simon Furst and Markus Bechter. AUTOSAR for Connected and Autonomous Vehicles: The AUTOSAR Adaptive Platform. *Proceedings - 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN-W 2016*, pages 215–217, 2016.

[16] GENIVI. vsomeip in 10 minutes. https://github.com/GENIVI/vsomeip/wiki/vsomeip-in-10-minutes, March 2018. [Online; accessed 3-July-2020].

[17] GENIVI. Commonapi c++ user guide. https://docs.projects.genivi.org/ipc.common-api-tools/3.1.3/html/CommonAPICppUserGuide.html, July 2019. [Online; accessed 3-July-2020].

[18] Jesse Howarth. Why discord is switching from go to rust. https://blog.discord.com/why-discord-is-switching-from-go-to-rust-a190bbca2b1f, February 2020. [Online; accessed 24-June-2020].

[19] IEEE Computer Society. *IEEE Standard for Local and metropolitan area networks–Time and Synchronization for Time-Sensitive Applications in Bridged Local Area Networks*. Number March 30. 2011.

[20] IEEE Standards Association. *IEEE Standard for Local and metropolitan area networks–Bridges and Bridged Networks–Amendment 28: Per-Stream Filtering and Policing*. 2017.

[21] Alexandru Kampmann, Andreas Wustenberg, Bassam Alrifaee, and Stefan Kowalewski. A Portable Implementation of the Real-Time Publish-Subscribe Protocol for Microcontrollers in Distributed Robotic Applications. *2019 IEEE Intelligent Transportation Systems Conference, ITSC 2019*, pages 443–448, 2019.

[22] Amit Levy, Bradford Campbell, Branden Ghena, Pat Pannuto, Prabal Dutta, and Philip Levis. The case for writing a kernel in rust. *Proceedings of the 8th Asia-Pacific Workshop on Systems, APSys 2017*, 2017.

[23] Amit Levy, Daniel B. Giffin, Bradford Campbell, Pat Pannuto, Philip Levis, Branden Ghena, and Prabal Dutta. Multiprogramming a 64 kB Computer Safely and Efficiently. *SOSP 2017 - Proceedings of the 26th ACM Symposium on Operating Systems Principles*, pages 234–251, 2017.

[24] L A N Man, Standards Committee, and Ieee Computer. *IEEE Standard for Local and metropolitan area networks–Frame Replication and Elimination for Reliability*. 2017.

[25] Nicholas Matsakis and Felix S. Klock. The rust language. *HILT 2014 - Proceedings of the ACM Conference on High Integrity Language Technology*, page 103, 2014.

[26] Ahmed Nasrallah, Akhilesh S. Thyagaturu, Ziyad Alharbi, Cuixiang Wang, Xing Shao, Martin Reisslein, and Hesham ElBakoury. Ultra-low latency (ULL) networks: The IEEE TSN and IETF DetNet standards and related 5G Ull research. *IEEE Communications Surveys and Tutorials*, 21(1):88–145, 2019.

[27] Object Management Group. Remote Procedure Call over DDS. pages 1–173, 2017. [Online; accessed 22-June-2020].

[28] OMG. Data Distribution Service. (April):1–180, 2015. [Online; accessed 22-June-2020].

[29] Stack Overflow. Stack overflow developer survey. https://insights.stackoverflow.com/survey/2016#technology-most-loved-dreaded-and-wanted, 2016. [Online; accessed 23-June-2020].

[30] Stack Overflow. Stack overflow developer survey. https://insights.stackoverflow.com/survey/2017#most-loved-dreaded-and-wanted, 2017. [Online; accessed 23-June-2020].

[31] Stack Overflow. Stack overflow developer survey. https://insights.stackoverflow.com/survey/2018/#most-loved-dreaded-and-wanted, 2018. [Online; accessed 23-June-2020].

[32] Stack Overflow. Stack overflow developer survey. https://insights.stackoverflow.com/survey/2019#technology-_-most-loved-dreaded-and-wanted-languages, 2019. [Online; accessed 23-June-2020].

[33] Stack Overflow. Stack overflow developer survey. https://insights.stackoverflow.com/survey/2020#technology-most-loved-dreaded-and-wanted-languages-loved, 2020. [Online; accessed 23-June-2020].

[34] Mike P. Papazoglou and Willem Jan Van Den Heuvel. Service oriented architectures: Approaches, technologies and research issues. *VLDB Journal*, 16(3):389–415, 2007.

[35] Navrattan Parmar, Virender Ranga, and B Simhachalam Naidu. Syntactic Interoperability in Real-Time Systems, ROS 2, and Adaptive AUTOSAR Using Data Distribution Services: An Approach. In G Ranganathan, Joy Chen, and Álvaro Rocha, editors, *Inventive Communication and Computational Technologies*, pages 257–274, Singapore, 2020. Springer Singapore.

[36] Andre Pinho, Luis Couto, and Jose Oliveira. Towards rust for critical systems. *Proceedings - 2019 IEEE 30th International Symposium on Software Reliability Engineering Workshops, ISSREW 2019*, pages 19–24, 2019.

[37] Graham Smethurst. GENIVI: Changing the In-Vehicle Infotainment Landscape. page 14, 2010. [Online; accessed 20-June-2020].

[38] IEEE Spectrum. Interactive: The top programming languages 2016. https://spectrum.ieee.org/static/interactive-the-top-programming-languages-2016, 2016. [Online; accessed 24-June-2020].

[39] Bechmarksgame Team. Which programming language is fastest? https://benchmarksgame-team.pages.debian.net/benchmarksgame/, 2020. [Online; accessed 24-June-2020].

[40] MSRC Team. A proactive approach to more secure code. https://msrc-blog.microsoft.com/2019/07/16/a-proactive-approach-to-more-secure-code/, July 2019. [Online; accessed 23-June-2020].

[41] MSRC Team. Why rust for safe systems programming. https://msrc-blog.microsoft.com/2019/07/22/why-rust-for-safe-systems-programming/, July 2019. [Online; accessed 23-June-2020].

[42] Lars Völker. Interoperability from AUTOSAR to GENIVI. 2011. [Online; accessed 20-June-2020].

[43] Lars Völker. Scalable service-oriented middleware over ip (some/ip). some-ip.com, 2020. [Online; accessed 21-June-2020].

[44] Feng Wang, Fu Song, Min Zhang, Xiaoran Zhu, and Jun Zhang. KRust: A formal executable semantics of rust. *Proceedings - 2018 12th International Symposium on Theoretical Aspects of Software Engineering, TASE 2018*, 2018-January:44–51, 2018.

[45] Automotive Wiki. Basic software module — automotive wiki,. https://automotive.wiki/index.php?title=Basic_Software_Module&oldid=2094, 2017. [Online; accessed 16-June-2020].

[46] Britta Wuelfing. CeBIT 2009: BMW and Partners Found GENIVI Open Source Platform. https://www.linuxpromagazine.com/Online/News/CeBIT-2009-BMW-and-Partners-Found-GENIVI-Open-Source-Platform, 2009. [Online; accessed 20-June-2020].