Master Thesis

Julius-Maximilians-
UNIVERSITÄT
WÜRZBURG

Investigation of the applicability of Industry 4.0 communication in vehicles

Ramadan Shweiki
Department of Computer Science
Chair of Computer Science II (Software Engineering)

Prof. Dr.-Ing. Samuel Kounev
First Reviewer

Prof. Dr. Tobias Hoßfeld
Second Reviewer

Stefan Herrnleben, M.Sc.
First Advisor

In the context of an industrial research project in cooperation with **Intedis GmbH & Co. KG**

Submission
28. April 2021

www.uni-wuerzburg.de

Abstract

In-vehicle communication became complex system of a reasonable number of distributed *Electronic Control Units (ECUs)* with even more sensors and actuators attached. The amount of data that needs to be transmitted within a vehicle has increased tremendously. Traditional communication bus based systems are difficult to meet today's requirements. Additional network bandwidth is mandatory to cover new functional requirements and cannot be met with traditional bus systems such as the *Controller Area Network (CAN)*. Low bandwidth from one side and increasing number of ECUs and their data from other side are downgrading the system a little bit, that is, the current infrastructure cannot meet the new trends such as autonomous driving and advanced driving assistance systems. In CAN bus, various devices have to be configured manually, which takes a lot of time and involves a high probability of errors. Therefore, automated solutions are needed. Even though ECUs are often vulnerable devices with many security flaws, in-vehicle networks used to be an isolated, hard-to-reach attack surface, which makes cars unattractive targets to hackers. In order to do justice to the complexity of a modern automobile, manufacturers are increasingly using on standardized platforms such as *AUTomotive Open System ARchitecture (AUTOSAR)*, which abstract the overall system from the hardware used and modularize components. *Scalable service-Oriented MiddlewarE over IP (SOME/IP)* have been designed by AUTOSAR but it is not used properly until today.

This thesis investigates the applicability of Industry 4.0 communication protocol called *Open Platform Communications Unified Architecture (OPC UA)* in vehicles. OPC UA is a service oriented machine-to-machine communication protocol and it's one of the most common important protocols in the area of industrial automation and is also partly used in the area of *Internet of things (IoT)*. OPC UA has its strength in the semantic modeling of information and is currently the only publish/subscribe protocol in the Industry 4.0 and IoT environment that supports it. The semantic model in OPC UA, called *information model*, provides an standardized message format, which facilitates the interoperability of devices from different manufacturers and enables more efficient and autonomous interaction between systems. For that, we chose a complex communication scenario in-vehicle communication called keyless entry and implemented it in OPC UA as prototype application. Keyless entry is term for an automotive technology which allows a driver to lock and unlock a vehicle without active use of the key. With that we can investigate the extent to which the OPC UA communication protocol is also suitable for the communication of sensors and actuators in the vehicle.

In this thesis we gathered information about OPC UA from current researches to understand the principles of technologies in order to configure the system. In addition, the theoretical comparison of OPC UA with SOME/IP and other *Message Oriented Middleware (MOM)* protocols supporting *publish/subscribe (PubSub)* on the theoretical level such as licensing, real-time, *Quality of Service (QoS)* and security. We developed two testbeds. The first testbed compare the two communication modes client/server and PubSub, which are both supported by OPC UA. The second testbed is the implementation of the keyless

entry use case. We selected open source OPC UA open62541 stack for our implementation. Our keyless entry application consists of five important components, implemented with OPC UA PubSub. We measure the memory and CPU consumption of both testbeds to show, whether they are high or low and feasible for time uncritical applications. We also identify a suitable structure and representation of exchanged data. We created an information model provided by the OPC UA specifications to describe the meaning of keyless entry instances.

The proof of concept implementation of the complex keyless entry use case shows that OPC UA meets the requirements for in-vehicle communication from the functional side. We evaluated our testbeds in terms of resource requirements and consumption (CPU, RAM). In comparison to the other MOM protocols, OPC UA is the only protocol that supports semantic data, which makes it well suited for IoT applications and in-vehicle communication. The contributions of this thesis can be summarized in three parts. The first contribution is how we can model the keyless entry use case with OPC UA. We considered which entities would be involved and how to represent the communication between entities with OPC UA. As second, we designed information model, how keyless entry can be structured in semantic data. It is about how the messages should look and what kind of messages are exchanged. The third contribution is the evaluation and implementation of prototypes whereby several performance measurements can be performed. In addition, we investigated the feasibility of the keyless entry use case with OPC UA.

The open62541 OPC UA PubSub implementation is faster than client/server for almost all package sizes. Based on our results, the keyless entry application can run on resource limited devices and maximize energy efficiency by using PubSub with UDP as transport protocol with low message overhead. OPC UA supports the wake-up via network for energy efficiency. The use of OPC UA PubSub with UDP for our use case is well suited for *Time-Sensitive Networking (TSN)* networks and devices with constrained resources like sensor nodes and low power processors. In addition, OPC UA is also real-time capable through the combination with TSN communication.

Zusammenfassung

Die Kommunikation im Fahrzeug wurde zu einem komplexen System aus einer größeren Anzahl von verteilten elektronischen Steuergeräten *Electronic Control Units (ECUs)* mit noch mehr angeschlossenen Sensoren und Aktuatoren. Die Menge der Daten, die innerhalb eines Fahrzeugs übertragen werden müssen, ist enorm gestiegen. Traditionelle, auf Kommunikationsbussen basierende Systeme, können den heutigen Anforderungen nur schwer gerecht werden. Zusätzliche Bandbreite in den Netzwerken der Fahrzeuge ist zwingend erforderlich, um neue Funktionsanforderungen abzudecken, die mit traditionellen Bussystemen wie *Controller Area Network (CAN)* nicht erfüllt werden können. Geringe Bandbreiten auf der einen Seite und eine steigende Anzahl von ECU und deren Daten auf der anderen Seite führen dazu, dass die derzeitige Infrastruktur den neuen Trends wie dem autonomen Fahren und fortschrittlichen Fahrassistenzsystemen nicht gerecht wird. Im CAN-Bus müssen verschiedene Geräte manuell konfiguriert werden, was sehr zeitaufwendig und mit einer hohen Fehlerwahrscheinlichkeit verbunden ist. Deshalb werden automatisierte Lösungen benötigt. Obwohl ECUs oft anfällige Geräte mit vielen Sicherheitslücken sind, waren bordeigene Netzwerke bisher eine isolierte, schwer zugängliche Angriffsfläche, was Autos zu unattraktiven Zielen für Angreifer macht. Um der Komplexität eines modernen Automobils gerecht zu werden, setzen die Hersteller zunehmend auf standardisierte Plattformen wie *AUTomotive Open System ARchitecture (AUTOSAR)*, die das Gesamtsystem von der verwendeten Hardware abstrahieren und Komponenten modularisieren. *Scalable service-Oriented MiddlewarE over IP (SOME/IP)* ist von AUTOSAR entworfen worden, wird aber bis heute nicht richtig genutzt.

In dieser Masterarbeit untersuchen wir die Anwendbarkeit des Industrie 4.0 Kommunikationsprotokolls *Open Platform Communications Unified Architecture (OPC UA)* in Fahrzeugen. OPC UA ist ein serviceorientiertes Kommunikationsprotokoll, welches für den Datenaustausch zwischen Maschinen konzipiert wurde. OPC UA gehört zu den wichtigsten Protokollen im Bereich der industriellen Automatisierung und wird auch verwendet im Bereich des *Internet of things (IoT)*. OPC UA hat seine Stärke in der semantischen Modellierung von Informationen und ist derzeit das einzige *publish/subscribe (PubSub)*-Protocol im Industrie 4.0- und IoT-Umfeld, das dies unterstützt. Das semantische Modell in OPC UA, genannt *Informationsmodell*, bietet ein standardisiertes Nachrichtenformat, das die Interoperabilität von Geräten verschiedener Hersteller erleichtert und eine effizientere und autonomere Interaktion zwischen Systemen ermöglicht. Dazu haben wir ein komplexes Kommunikationsszenario im Fahrzeug, das sogenannte Keyless Entry, ausgewählt und in OPC UA als Prototypanwendung entwickelt. Keyless Entry ist die Bezeichnung für eine Automobiltechnologie, die es dem Fahrer ermöglicht, ein Fahrzeug ohne aktive Benutzung des Schlüssels zu verriegeln und zu entriegeln. Damit untersuchen wir, inwieweit das Kommunikationsprotokoll OPC UA auch für die Kommunikation von Sensoren und Aktuatoren im Fahrzeug geeignet ist.

In dieser Arbeit haben wir Informationen über OPC UA aus aktuellen Forschungen gesammelt, um die Prinzipien der Technologien zu verstehen, damit das System konfiguriert

werden kann. Außerdem der theoretische Vergleich von OPC UA mit SOME/IP und anderen *Message Oriented Middleware (MOM)* Protokollen, die PubSub unterstützen, auf der theoretischen Ebene wie Lizenzierung, Echtzeit, *Quality of Service (QoS)* und Sicherheit. Wir entwickelten zwei Testumgebungen. In der ersten Testumgebung wollen wir die beiden Kommunikationsmodi von OPC UA client/server mit PubSub vergleichen und bewerten. Die zweite Testumgebung ist die Implementierung des Anwendungsfalls Keyless Entry. Für die Testumgebungen haben wir den Open Source OPC UA Stack open62541 gewählt. Unsere Keyless Entry Applikation besteht aus fünf wichtigen Komponenten, die mit OPC UA PubSub implementiert wurden. Wir wollen den Speicher und CPU Verbrauch beider Testumgebungen messen, um zu zeigen, ob sie hoch oder niedrig und für zeitunkritische Anwendungen praktikabel sind. Wir identifizieren auch eine geeignete Struktur und Darstellung der ausgetauschten Daten. Wir haben ein Informationsmodell erstellt, das von den OPC UA Spezifikationen bereitgestellt wird, um die Bedeutung von Keyless Entry Instanzen zu beschreiben.

Die Machbarkeitsstudie für den komplexen Anwendungsfall Keyless Entry zeigt, dass OPC UA die Anforderungen an die Kommunikation im Fahrzeug von der funktionalen Seite her erfüllt. Wir haben unsere Testumgebungen im Hinblick auf Ressourcenbedarf und -verbrauch (CPU, RAM) ausgewertet. Im Vergleich zu den anderen MOM Protokollen ist OPC UA das einzige Protokoll, das semantische Daten unterstützt, wodurch es sich gut für IoT-Anwendungen und die Kommunikation im Fahrzeug eignet. Die Beiträge dieser Arbeit lassen sich in drei Teilen zusammenfassen. Der erste Beitrag ist, wie wir den Anwendungsfall des Keyless Entry mit OPC UA modellieren können. Wir haben uns überlegt, welche Entitäten beteiligt sein werden und wie die Kommunikation zwischen den Entitäten mit OPC UA abgebildet werden kann. Als zweites haben wir ein Informationsmodell entworfen, wie Keyless Entry in semantischen Daten strukturiert werden kann. Es geht darum, wie die Nachrichten aussehen sollen und welche Art von Nachrichten ausgetauscht werden. Als drittes haben wir eine Evaluation durchgeführt und Prototypen implementiert, mit denen verschiedene Leistungsmessungen durchgeführt werden können. Außerdem haben wir die Anwendbarkeit des Anwendungsfalls Keyless Entry mit OPC UA untersucht.

Die open62541 OPC UA PubSub ist schneller als client/server für fast alle Paketgrößen. Basierend auf unseren Ergebnissen kann die Keyless Entry Applikation auf ressourcenbeschränkten Geräten funktionieren und die Energieeffizienz maximieren, indem PubSub mit UDP als Transportprotokoll mit geringem Nachrichten-Overhead verwendet wird. OPC UA unterstützt das Aufwecken über das Netzwerk zur Energieeffizienz. Die Verwendung von OPC UA PubSub mit UDP für unseren Anwendungsfall ist gut geeignet für *Time-Sensitive Networking (TSN)* Netzwerke und Geräte mit eingeschränkten Ressourcen wie Sensoren und Prozessoren mit geringer Leistung. Darüber hinaus ist OPC UA durch die Kombination mit der TSN-Kommunikation auch echtzeitfähig.

Contents

1. Introduction

Automotive systems became complex systems of a reasonable number of distributed *Electronic Control Units (ECUs)* with even more sensors and actuators attached. In today's cars the number of ECUs reached 70, processing about 2500 signal points and their numbers are still growing [1]. Cars can be considered as computer networks on wheels. The nodes in these networks are typically ECUs responsible for various car features such as engine control, brakes and also comfort functions such as keyless vehicle access, light assistant and multimedia [2]. With the development of advanced driving assistance systems, the amount of data that needs to be transmitted within a car has increased tremendously. Traditional communication bus-based systems are unable to meet today's requirements [3]. Automotive networks are essential for both driver assistance and future trends such as autonomous driving.

Both the increasing number of sensors and the desire for more flexibility poses new challenges for communication within the vehicle. Low bandwidth from one side and increasing number of ECUs and their data from other side are downgrading the system a little bit, that is, the current infrastructure cannot meet the new technology advancement, such as remote diagnostics, high bandwidth applications, applications in need of more shared resources [4]. In-vehicle communication makes special demands on the communication middleware. On the one hand, a variety of communication paradigms are used for in-vehicle communication, such as signal-based and function-based communication. On the other hand, the communication partners differ considerably in terms of computing resources and complexity of the hosted applications. Various Ethernet-based protocols were supposed to unify the communication in the first decade of the new century, but the effort was only partially successful [5]. In CAN-Bus various devices must be configured manually requires very much time and involves a high probability of error. In addition, the entire system is statically configured and tested, so that adding or changing a component currently requires total integration. Greater flexibility with regard to the software components in the vehicle makes sense in order to reduce the enormous effort involved in the overall integration [6, 7]. With increasing complexity and rising numbers of network devices, the possible impact of malicious manipulation and malfunction also increases [8]. Even though ECUs are often vulnerable devices with many security flaws, intra-vehicular networks used to be an isolated, hard-to-reach attack surface, which makes cars unattractive targets to hackers.

1.1 State of the Art

In a traditional vehicle, sensors and actuators are largely networked via bus systems such as CAN-Bus, *Local Interconnect Network (LIN)*, *Media Oriented Systems Transport (MOST)*, or *FlexRay*. They are based on a signal-oriented approach, where the sender decides whether data should be transmitted or not independent of a receivers request. Consequently, a significant amount of non-requested data is occupying the bandwidth which may cause internal communication problems.

The automotive embedded system industry tried to create a universal, scalable, low cost, high performance system, by the usage of Ethernet [9]. It didn't fulfils the identified requirements for in-vehicle communication but the usage of Ethernet to connect heterogeneous ECUs created a platform in which *service-oriented architecture (SOA)* can be implemented. SOA protocols designed for high level computers and servers cannot be used in automotive embedded system. These upcoming challenges can be tackled by the introduction of *Service-Oriented Communication (SOC)* protocols into the vehicle's communication system. So new protocol/middleware named SOME/IP have been designed by AUTOSAR [10]. It is the first standard for IP-based middleware solution using a different approach, namely *Service-Oriented Communication (SOC)*.

In contrast to signal-oriented systems, service-oriented systems communicate data exclusively on the receivers demand and not when a sender finds a transmission appropriate, hence an increase in qualitative bit rate. Furthermore, SOME/IP's network traffic is much more dynamic and unpredictable compared to classic automotive protocols like CAN because of its service-oriented nature. Although SOME/IP deemed very promising as a communication middleware, but it does not include any security functionality and other disadvantages which may impose unwanted limitations and introduce excessive overhead, that's why it was rarely used [11]. No existing IP-based middleware fulfils the identified requirements for in-vehicle communication. So the problem is not solved and the challenge increases with more sensors and their huge amount of data. However, today's most used automotive communication bus is still the CAN-Bus [12].

1.2 Idea

The aim of this work is to investigate the applicability of the *Open Platform Communications Unified Architecture (OPC UA)* communication protocol for in-vehicle communication. We investigate a complex use case to show the applicability in this case. From this investigation we can then make an educated guess, that OPC UA seems to be suitable for in-vehicle communication. We compare OPC UA with SOME/IP from current researches to understand the principles of technologies in order to configure in-vehicle communication network and implement representation vehicular communication use case.

The first objective of this thesis is the specification of complex use case in-vehicle communication by modeling its entities and the information that is exchanged between the entities. We describe the use case with the exact description of all its components individually and how they all interact with each other. When selecting and applying a complex use case with OPC UA, we can assume that OPC UA is also applicable for other use cases and therefore applicable in-vehicle communication. A vehicular communication use case is represented, the requirement and the entities for communication within the use case are specified. Then as second objective is to identify a suitable structure and representation of exchanged data. We identify an information model to structure the use case in semantic data to describe the meaning of use case instances and the representation of exchanged

data. Without information modelling of the use case it will be difficult to understand the implementation of the use case. This is also important to ensure a good definition and understanding of the complex use case. The third objective is the implementation of the in-vehicle communication use case. We want to proof that the implementation of in-vehicle communication use case is possible by using OPC UA. We develop prototypes to evaluate the performance measurements (RAM, CPU) and network latency to show wether OPC UA is suitable for limited-resource devices so that developers can size hardware environments to minimize costs while providing an appropriate performance. In this work we look at the resource consumption on the server and on the client. OPC UA is analyzed from the aspect of stack performance such as measure time to execution on request and protocol complexity or header. Evaluation of the response time measurements for OPC UA messages in-vehicle communication network and check whether it is high or low and feasible for time critical requirements.

For a high level of abstraction in communication between individual software components, OPC UA is used in this work, which enables the dynamic establishment of communication channels between the components during system runtime (service discovery). We want to know whether to wake up servers via network is possible and check whether the authentication of the service client or other security measures is possible in OPC UA. Furthermore, it is assumed that we have real-time requirement for our application. Real-time is generally not supported in the OPC UA PubSub but PubSub with *Time-Sensitive Networking (TSN)* technology has become more efficient and real-time capable.

1.3 Benefit

The feasibility of a complex use case in the area of in-vehicle communication with OPC UA, states that other applications for in-vehicle communication can also be applied with OPC UA. It follows that the implementation of OPC UA over in-vehicle communication is possible. With new and improved real-time features added to OPC UA protocols over the last few years, which allows the deployment of OPC UA in time critical applications in vehicles and it is therefore capable for real-time communication. OPC UA supports semantic modeling of information, which makes it a device centric protocol that focuses on device interoperability. Benefits of interoperability of sensors and actuators from different manufacturers is reflected in higher efficiency and further acceleration of in-vehicle communication and emphasis on a secure transfer of data.

Using OPC UA PubSub promises for servicing a large number of clients because it reduces both CPU use and the network load, which leads to an increase the number of sensors and therefore meets today's requirements in-vehicle communication. In addition, if OPC UA is applicable in-vehicle communication, which it also supports wake-up of devices via network, that allows for small devices to go back to sleep without waiting for a response from other devices. This reduces the communication overhead to increase the battery lifetime and maximize energy efficiency.

1.4 Contributions and Research Questions

The contributions of this thesis can be summarized in three parts. At first, we model the keyless entry use case with OPC UA. We consider which entities would be involved and how to represent the communication between entities with OPC UA. A second contribution of this work is the structuring of keyless entry use case in semantic data by designing an information model. As third, we perform evaluation and implemented prototypes whereby several performance measurements can be performed. In addition, we

investigate the feasibility of the keyless entry use case with OPC UA. In detail we pursue the following goals and defined the following research questions. These research questions will be answered in the related sections in this thesis.

Goal 1 Exploration of the current status of communication within vehicles.

- **RQ 1.1** Which network protocol requirements can be specified for communication within a vehicle?
- **RQ 1.2** How are these requirements fulfilled at the current stage of development?

Goal 2 Investigation of the functional features of OPC UA for inner vehicle use.

- **RQ 2.1** Which features are provided by OPC UA per default?
- **RQ 2.2** Which requirements can be met by OPC UA?
- **RQ 2.3** In which areas can OPC UA only be used in a limited way?
- **RQ 2.4** In which areas can OPC UA not be used due to non-fulfillable requirements?

Goal 3 Specification of representative vehicular communication use case.

- **RQ 3.1** Which representative use case is suitable to investigate the protocol characteristics?
- **RQ 3.2** What are the requirements for this use case?
- **RQ 3.3** Which entities have to communicate with each other in this use case?
- **RQ 3.4** Which information have to be exchanged at the specified use case?

Goal 4 Identifying a suitable structure and representation of exchanged data.

- **RQ 4.1** What is the data type of the exchanged information and how can it be structured?
- **RQ 4.2** How can the information be mapped to an existing information model?
- **RQ 4.3** How could an information model, dedicated to the investigated use case, look like? *(optional)*

Goal 5 Implementation of representative vehicular communication use case.

- **RQ 5.1** Which implementation (simulation, emulation, hardware) is suitable for the investigation of the use case?
- **RQ 5.2** How can the use case be represented in software?
- **RQ 5.3** Which real hardware can be integrated into the use case implementation? *(optional)*

Goal 6 Evaluation of the use case in terms of resource requirements.

- **RQ 6.1** What statements can be made about the local resource consumption (CPU, RAM) of OPC UA?
- **RQ 6.2** What statements can be made about communication with OPC UA?
- **RQ 6.3** What can be deduced about the suitability of OPC UA within vehicles?

1.5 Outline

The structure for the remainder of this thesis is organized as follows. Chapter 2 elaborates the theoretical knowledge for a better understanding of this thesis. Chapter 3 presents other works, which are related to in-vehicle communication network and discusses some of the existing tools and implementations regarding the OPC UA technology. Chapter 4 introduces the architecture of the use case and define a suitable structure and representation of exchanged data with the creation of an information model. Chapter 5 presents the implementation of representative in-vehicle communication use case using OPC UA implementation stack open62541 as communication protocol for the communication between components of this use case. Chapter 6 evaluates the keyless entry use case in terms of resource requirements and investigates the communication latency by exchanging data with OPC UA PubSub and client/server. Chapter 7 concludes this thesis and identifies further research aspects that could be considered in the future based on this work.

2. Background

This chapter provides the background knowledge that helps to better understand the work. Section 2.1 describes an existing IP-based protocol SOME/IP. Section 2.2 defines important basics about OPC UA, describes OPC UA publish/subscribe communication mode and explains the information model with the graphical notation for specific application. An overview of in-vehicle communication networks is given in Section 2.3. Section 2.4 provides the background knowledge of electronic locking and starting system Keyless Entry/Go in vehicle.

2.1 SOME/IP

Scalable service-Oriented MiddlewarE over IP (SOME/IP) is open source licensing model designed by BMW group in 2011 and meanwhile is maintained by the AUTOSAR organization [13]. It is based on Ethernet and is located in ISO - OSI layers 5 to 7. In order to develop control devices that use this interface, a manufacturer must also have test benches that are compatible with Ethernet and SOME/IP. With SOME/IP it is also possible in systems to dynamically subscribe to services at runtime. SOME/IP supports a wide range of features including serialization, RPC, SD, PubSub and segmentation of UDP datagrams [14].

SOME/IP introduces a philosophical shift in automotive data transmission. All of the prior standards and protocols have been signal-oriented. In contrast, SOME/IP introduces service-oriented transmission of information. In service-oriented communication data is transmitted only if it is needed by at least one receiver, while in signal-oriented communication data is sent whenever the sender finds it necessary, for example when a value is updated. The obvious benefit of service-oriented communication is that it avoids unnecessary data flooding of the channel. However, this assumes that a server (i.e., the provider of the data) is informed about whether any receiver needs the data [15]. The advantage in comparison to other protocols is that needed data is only sent from the host to the client when the client is subscribed to the service. For communication some standard components from the Ethernet stack are used. In Figure 2.1 the location of the SOME/IP (-SD) protocol in the OSI model is shown. SOME/IP (-SD) is not a real time protocol because it is only based on standard Ethernet layers. Actually the real-time requirements are met by a moderate covered QoS. While this section describes the technical part of SOME/IP, Chapter 3 focuses on the concept of SOME/IP.

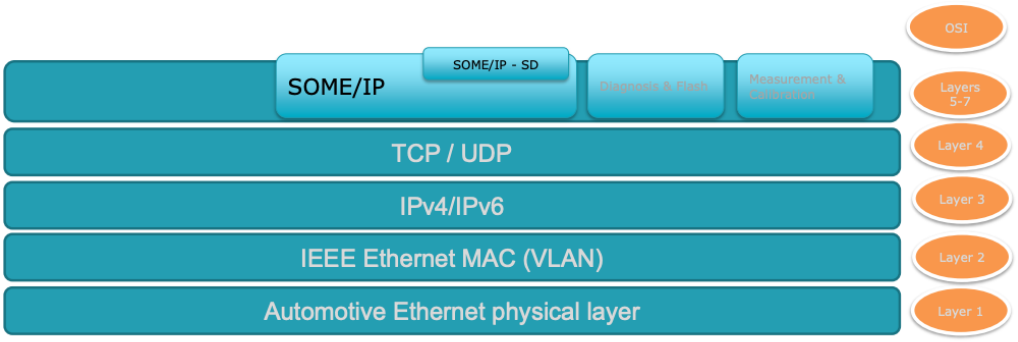

Figure 2.1: OSI model and classification of protocols [16].

SOME/IP defines a payload and a header, which contains two parts header and payload. The structure of the SOME/IP protocol can be seen in Figure 2.2.

- *Service ID*: unique identifier for each service.
- *Method ID*: 0-32767 for methods, 32768-65535 for events.
- *Length*: length of payload in byte (covers also the next IDs, that means 8 additional bytes).
- *Client ID*: unique identifier for the calling client inside the ECU has to be unique in the overall vehicle.
- *Session ID*: identifier for session handling has to be incremented for each call.
- *Protocol Version*: 0x01.
- *Interface Version*: major version of the service interface.
- *Message Type*: (8 bit).
- *Return Code*: (8 bit).

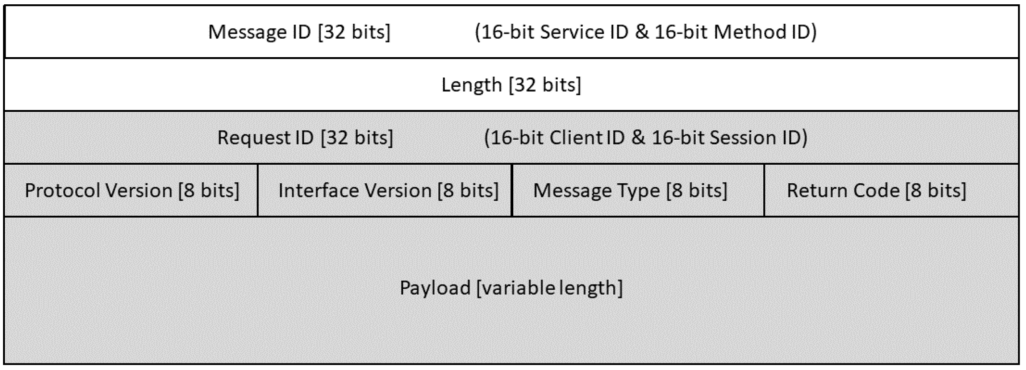

Figure 2.2: The structure of the SOME/IP protocol [17].

2.2 OPC UA

OPC UA, released in 2008, is an open platform independent SOA for communication of industrial automation devices and systems. It was developed to overcome the limitations presented by its predecessor, ”OPC Classic”. OPC UA is even used in a lot of areas where

it was not designed for, and there are many more areas where manufacturers want to use a standard like it [18].

The biggest difference to previous versions OPC Classic, is that machine data can not only be transported, but also semantically described in a machine-readable manner. For information exchange OPC UA offers two communication mechanisms:

- The first and most common architectural mode of OPC UA is a client-server model for industrial use cases without real-time requirements. The client accesses the information provided by the server through defined services [19].
- The second method of communication is OPC UA PubSub[1]. The new PubSub extension of OPC UA adds the possibility of many-to-many communication based on the *publish/subscribe (PubSub)* paradigm [20].

The PubSub communication pattern operates in contrast to OPC UA's conventional client-server layout, between publisher and subscriber. In the standard OPC UA client-server layout, sessions are used to maintain communication between a single client and server. Client-server subscriptions are confined to a single session, and therefore not shared between clients. This has been a major impedance on interoperability, which led the OPC UA Foundation to develop a second communication model [21]. In PubSub, requests and responses are never exchanged directly, and no communication context is maintained. There applications act as publishers and/or subscribers without any knowledge of other publishers/subscribers. All communication is between application and MOM, that act as either brokerless or broker-based form to forward messages.

OPC UA is a sophisticated, scalable and flexible mechanism for establishing secure connections between client and servers. Feature of this unique architecture include[2]:

Scalability: OPC UA is scalable and platform-independent. It can be supported on high-end servers and on low-end sensors.

A Flexible Address Space: The OPC UA Address Space is organized around the concept of an object. Objects are entities that consist of Variables and Methods and provide a standard way for servers to transfer information to clients.

Common Transports and Encodings: uses standard transports and encodings to ensure that connectivity can be easily achieved in both embedded and enterprise environments.

Security: OPC UA implements a sophisticated security model that ensures the authentication of client and servers, the authentication of users and the integrity of their communication.

Internet Capability: OPC UA is fully capable of moving data over the internet.

A Robust Set of Services: OPC UA provides a full suite of services for *Eventing*, *Alarming*, *Reading*, *Writing*, *Discovery* and more.

Certified Interoperability: OPC UA certifies profiles such that connectivity between a client and Server using a defined profile can be guaranteed.

A Sophisticated Information Model: OPC UA profiles more than just an object model. OPC UA is designed to connect objects in such a way that true information can be shared between clients and servers.

[1] `https://opcfoundation.org/developer-tools/specifications-unified-architecture/part-14-pubsub/`

[2] `https://www.rtautomation.com/technologies/opcua//`

Sophisticated Alarming and Event Management: UA provides a highly configurable mechanism for providing alarms and event notifications to interested clients. The *Alarming* and *Event* mechanisms go well beyond the standard change in-value type alarming found in most protocols.

Integration with Standard Industry Specific Data Models: The OPC Foundation is working with several industry trade groups that define specific information models for their industries to support those information models within UA.

OPC UA's strength lies in the address space, and this is what makes OPC UA a universal application interface. The use of UDP instead of TCP leads to a reduced header overhead and enables the use of *User Datagram Protocol (UDP)*/IPv6 header compression in *Low-Power Wireless Personal Area Networks (LoWPAN)*.

The security of OPC UA is based on mechanisms such as *Authentication*, *Authorization* and *Encryption*. There are three security options/levels available: *None*, *Sign* and *SignAndEncrypt*[3]. The Federal Office for Information Security (BSI) experts have intensively studied the specifications and stack implementations of OPC UA and concluded that OPC UA offers a high degree of security. All security methods are available. However, these methods must be implemented comprehensively and above all correctly by the users. The data modeling defines the rules and base building blocks necessary to expose an information model with OPC UA. The UA *Services* are the interface between servers as supplier of an information model and clients as consumers of that information model. The UA *Services* are the interface between servers as supplier of an information model and clients as consumers of that information model. Layer model of the OPC UA in Figure 2.3 is divided into two groups, infrastructure and information models. In the transport component two communications models are available: connection based client-server relationship and connectionless PubSub.

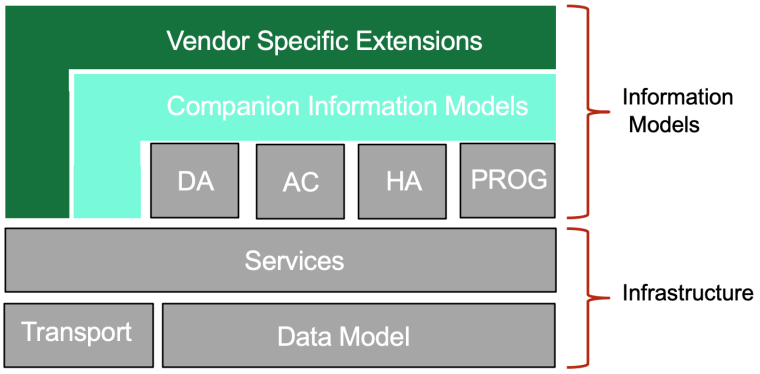

Figure 2.3: OPC UA layered architecture [22].

Information models for the domain of process information are defined by OPC UA on top of the base specifications, to cover all successful features known from Classic OPC. OPC Foundation collaborates with organizations and domain experts. Vendors could add specific features to cover their use cases.

The structure of the OPC UA specification consists of a thirteen-part specification, which is divided into three thematic areas and was published as IEC standard (IEC 62541). Figure 2.4 shows the structure of the OPC UA specification. Individual parts of the specification build on one another. The OPC Foundation therefore recommends working through

[3] `https://www.rfid-wiot-search.com/opc-ua-the-united-nations-of-automation/`

parts 1 to 5 in succession. The following sources were used to gather information used [23]. This section answers the research questions **R2.1** and **R2.2**, which are about the default features and requirements that are fulfilled with OPC UA. In addition, the most important features for OPC UA are illustrated again in Table 3.1.

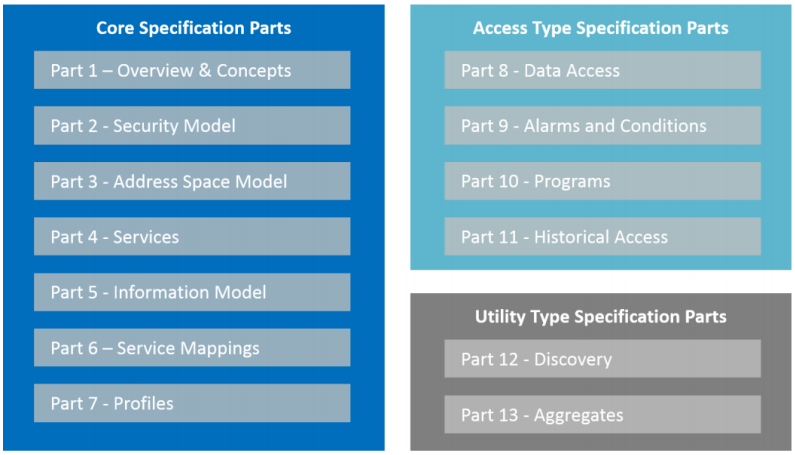

Figure 2.4: Structure of the OPC UA specification [24].

2.2.1 Publish/Subscribe in OPC UA

In contrast to the client-server model, the *publish/subscribe (PubSub)* method is not based on requests and responses, but on publishing and subscribing to information. A participant in a network can provide information. He does not know whether and how many other participants are interested in his information. Another participant can subscribe to information of interest to him without having to know about existing publishers. In OPC UA, the exchange of information takes place via a so-called MOM, so that the participants in the communication do not need to know about each other's existence. At the same time, the publisher has no influence on how many subscribers are interested in the information. Independence and scalability are thus achieved through PubSub [25].

PubSub can be used primarily for two types of networks. On the one hand, directly in a production environment where machines regularly exchange small data sets with each other and, on the other hand, in more complex networks where larger amounts of data are generated that are stored in a cloud, for example. UDP is suitable as a transport protocol for the first case, the *Advanced Message Queuing Protocol (AMQP)*or the *Message Queuing Telemetry Transport (MQTT)* protocol are suitable for the second case. A prerequisite for successful communication is a common understanding of data. A subscriber must be able to process the received data into information. Accordingly, the publisher must provide the data with the appropriate syntax and semantics. Figure 2.5 shows the actions of the publisher before a message is sent out.

Practically at the beginning of the chain is the available information, e.g. sensor values. From all the available information, the interesting information must first be information must first be filtered out. The decision as to which information is interesting is determined with the help of the *PublishedDataSet* parameters. The filtering process has produced data from the information *DataSet*. A syntax and semantics are now added to this data by the *DataSetWriter*, which is also supported by a configuration, the *DataSetMetaData*. This configuration must be publisher and subscriber so that the subscriber can process the incoming data. *DataSetMessages* are created. If necessary, these can be assigned to a

common *WriterGroup*, i.e. packed together in a message packet *NetworkMessage*. During this process, the security measures, e.g. the encryption and signature of the message package. The message can be forwarded to the MOM.

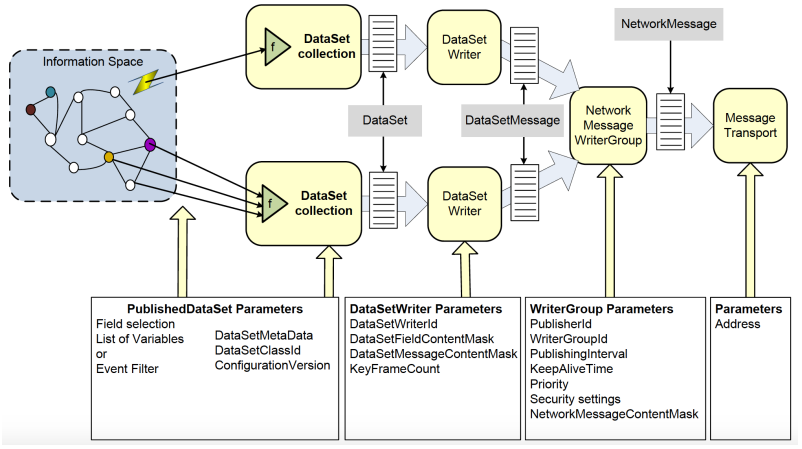

Figure 2.5: Publisher message sending sequence [25].

OPC UA does not define a MOM as itself, but only distinguishes between a brokerless and a broker-based middleware. The decision as to which variant of middleware is to be used is determined by the above mentioned use cases. Networks with small amounts of data vs. complex networks with large amounts of data.

The advantage of brokerless middleware is the elimination of extra software (a broker). The message packets can thus be sent directly from one participant via a switch or router to one or more other router to one or more other participants without being processed by an intermediate software. processed by intermediate software. The latency is kept low. UDP is suggested as the UDP is proposed as the communication protocol. Disadvantages of the use of broker-less middleware in connection with UDP are, however, an unguaranteed sequence of the of the packets or the loss of packets.

The broker-based middleware variant, on the other hand, offers other advantages. Through the the use of a broker, the communication participants are not in direct contact with each other. with each other. Publishers and subscribers do not need a uniform communication protocol, but can communicate with the broker via different protocols. The subscriber can subscribe to messages from the broker. The broker then filters incoming messages from from publishers and sends only the desired messages to the subscriber. Furthermore, messages can be temporarily stored by the broker. The publisher can send a message to the broker and switch to an inactive state. At a subscriber can still receive the publisher's messages through the broker. the broker.

The procedure for receiving messages from the subscriber is similar to the procedure for preparing to send messages from the publisher in Figure 2.5, but in reverse order. Incoming messages are filtered, if necessary (if not done by the broker) and desired messages are decrypted. By sharing *DataSetMetaData* with the publisher, the data can be decoded and information is produced that can be used by the subscriber.

In OPC UA PubSub there are four protocols that can be used to publish messages, AMQP, MQTT, OPC UA UDP, OPC UA Ethernet. OPC UA UDP uses UDP to transmit data. It has less overhead than other transport protocols e.g. TCP, because it doesn't establish connections between the communication entities. Those transport protocols must include

Message Mappings in their payload. *Message Mappings* describe how to structure and encode the messages. There are two of those mappings in the OPC UA PubSub. One is JSON and the other one is the *UA Datagram Protocol (UADP)*.

JSON is especially popular with web and enterprise software. It's built on a subset of JavaScript programming language. It employs human readable text, making it simple to read and write. It is dependent on the broker's security mechanisms. UADP focuses on a very efficient structure to fulfill the requirements of cyclic communication. It defines a *NetworkMessage* which can be included in the payload of a transport protocol [26].

2.2.2 OPC UA Information Model

Classic OPC transfers only minimal information to understand the semantic of the transmitted data. For example the tag name and the engineering unit are transmitted with the provided data. It is possible to define the semantics of the given data using OPC UA. For information modeling in OPC UA these conventions are fundamental [18]:

- Hierarchies and single inheritance are used in an object-oriented manner.
- Type information and instances shall be retrievable in the exact same manner.
- In a complete mesh network, nodes can be connected in a variety of ways to link information, giving it the flexibility to depict a variety of use cases.
- Class hierarchies and type of references can be specified to expand the functionality of OPC UA.
- The OPC Server is the only location where the OPC UA knowledge model is applied.

2.2.3 Information Model Graphical Notation

In this subsection we explain a graphical notation for modelling an OPC UA use case. OPC UA defines a graphical notation for modelling an OPC UA Address Space of the application (see 2.6). Generally speaking information models are sets of readymade types of objects, variables, references, events and data. The following is an explanation of the most important predefined *ReferenceTypes*, *ObjectTypes* and *VariableTypes*.

Hierarchical References are used to define hierarchies in the Address Space, though it does not preclude loops. It is an abstract *ReferenceType*.

Hierarchical References: HasProperty the target node describes properties of the source node. Mostly non variable properties of objects, variables and methods are displayed.

Hierarchical References: HasComponent the target node is part of the source node. Relationships of objects, variables and methods are shown.

Non Hierarchical Reference there are two types of it, symmetric and asymmetric *Reference*.

HasSubType is used to express subtype relationships in the type hierarchy. For example, abstract *ReferenceTypes* can have concrete subtypes using this *ReferenceType*. It is a concrete *ReferenceType*.

HasTypeDefinition binds *Objects* or *Variables* to their respective *ObjectTypes* or *VariableTypes*. It is a concrete *ReferenceType*.

ObjectType this class defines how a subsequent instance (object) should look like attributes, references, properties. *Objects* are always based on *ObjectTypes*.

Object class of a node which is an object a real-world object or system component. *Objects* have properties, variables, methods, type definitions, etc. All of these are linked with an object by references.

Variable there are two kinds of variables *Properties* and *DataVariables*. *Properties* characterize nodes. *DataVariables* form the content of objects. *DataVariables* can have *Properties*, but *Properties* cannot have any *DataVariables* or *Properties*.

VariableType nodes of this class provide type definitions for variables.

Method class of nodes that define callable functions.

DataType nodes of this class describe the syntax of a variable value.

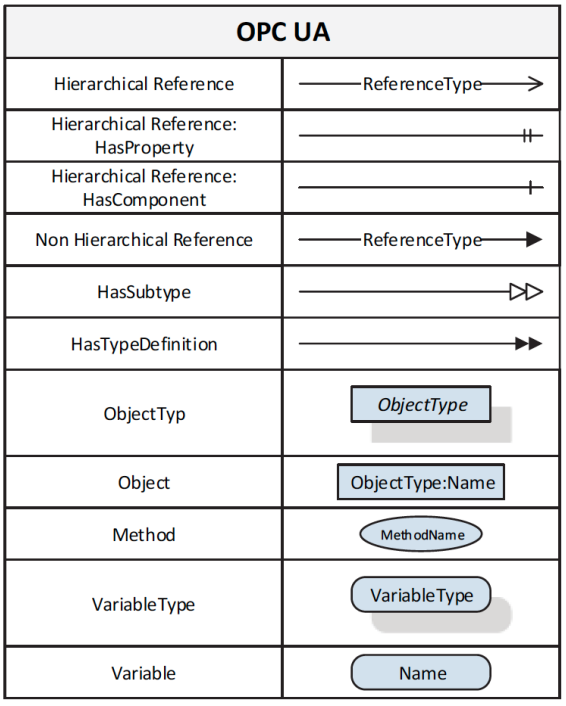

Figure 2.6: OPC UA information model graphical notation [27].

2.2.4 OPC UA Software Libraries and Tools

There exist several *Software Development Kit (SDK)*s for rapid development of OPC UA applications. Some of them are commercial, and some are open source. The commercial ones are offered by companies like Integration Objects, Unified Automation, National Instruments, or MathWorks. Components of the commercial applications usually come with additional documentation and professional support. All client and server open source OPC UA implementations in different programming languages are listed in the Table 2.1. They usually differ in the implemented functionality, ease of use, and quality of documentation. The majority of these software components do not provide all features as proposed in the OPC UA specification [28]. There are currently others open source implementations of OPC UA. All of them[4] are maintained and developed by serious companies and research institutes. Amongst some of the implementations, the ones worth mentioning are Open62541[5] are written in C programming language. Open62541 implementation is used for many purposes including [29]:

[4] `https://github.com/open62541/open62541/wiki/List-of-Open-Source-OPC-UA-Implementations`
[5] `https://open62541.org/doc/current/`

- It is open, constantly updated and very well documented on GitHub[6]. Compared to the other implementations this is much more convenient.
- It is very portable, capable of running on Windows, Linux, Android and some embedded systems.
- It includes a lot of already implemented examples on top of which one can build all sorts of communication scenarios.
- It is scalable, providing an event-based architecture.

Name	***Language***	***License***
*open*62541	*C*	$MPL - 2.0$
UA.NETStandard	*C#*	*GPL*
$node - opcua$	*JavaScript*	*MIT*
FreeOpcUa	$C++, Python$	*LGPL*
ASNeG	$C++$	*Apache*
EclipseMilo	*Java*	*EclipsePublicLicense*
S2OPC	*C*	*Apache*
opcua	*Rust*	$MPL - 2.0$

Table 2.1: Open source OPC UA client and server implementations available for commercial use.

OPC UA offers a lot of tools and great potential for creating information models, their use is not necessary. There are various tools which can be used to create *NodeSet2.xml* file. These are the most common open-source and commercial tools.

Unified Automation UaModeler UaModeler is a commercial tool which offers a nice GUI to define a custom nodeset. You can load other nodesets and extend them with a custom types and instances. After modeling the nodeset, you can export it to various formats, including the *NodeSet2.xml* format. The free version is limited to a maximum of 100 nodes.

Playing around with this tool revealed that for simple node sets, this tool can be a good starting point. If a node set involves a lot of inheritance and complex relations between nodes the UaModeler comes to its limitations.

Free OPC UA Modeler Another similar tool, but completely open-source, is the "Free OPC UA Modeler" (It is currently work in progress, but the current state looks quite promising). Free OPC UA Modeler is a tool for designing OPC UA address spaces. It uses OPC UA specified XML format which allows the produced XML to be imported into any OPC UA SDK [30].

UMX Pro – UA Model eXcelerator Professional The Beeond UMX Pro Tool is a graphical designing tool which can be used to design OPC UA Models similar to the UaModeler. This commercial tool also adds code generation capability for a number of SDK's (including Prosys Java, OPCF UANETStandard, Matrikon Flex, OPEN62541, and any other that can launch a CLI) [31].

[6] `https://github.com/open62541/open62541`

UA-ModelCompiler

The OPC Foundation's model compiler is used by various groups and by the OPC Foundation itself to create the official *NodeSet2.xml* files for the companion specifications. The compiler will generate C# and ANSI C source code from XML files which include the UA Services, data-types, error codes and numerous CSV files that contain NodeIds, error codes, and attributes. It is written in C#, but in combination with Mono, it can also be used on Linux [32].

One major drawback is, that this model compiler does not provide any GUI. You have to write your own Model.xml file manually using a text editor. The UA-ModelCompiler then reads this Model.xml file, checks its consistency and integrity, and then creates the *NodeSet2.xml* files, including the *Types.bsd*definition, and *NodeId.csv* files.

2.3 Vehicle Electrical System

Today's on-board communications network in vehicles is a heterogeneous computer network consisting of several CAN[7] for general signal-based communication, FlexRay[8] for real-time communication, MOST[9]for bandwidth-intensive multimedia applications and other networking technologies [33]. Since these technologies are not compatible, the Communication must be translated through an application layer gateway.

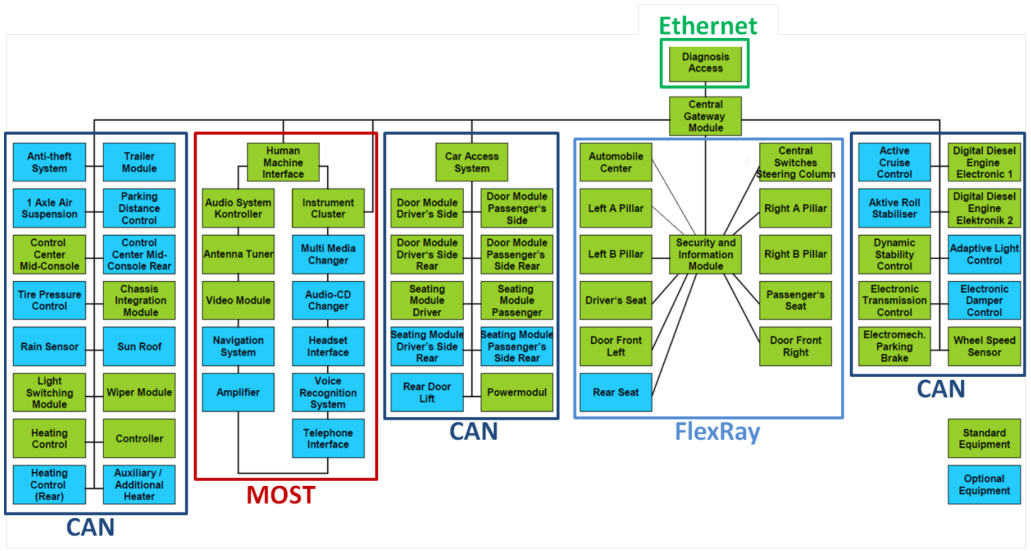

Figure 2.7: A simplified representation of today's vehicle electrical [34].

Figure 2.7 shows a simplified representation of a modern on-board communications network in the vehicle. There are more than 80 Electronic Control Unit (ECU) in a vehicle. ECUs are classified according to application areas and thus divided into different domains: Infotainment, telematics, HMI, body/comfort, driver assistance, power train and chassis are common application domains[35, 36]. This section and Section 2.1 answer the research questions **RQ 1.1 RQ 1.2** and therefore **Goal 1** one is fulfilled.

[7] `http://esd.cs.ucr.edu/webres/can20.pdf`
[8] `http://files.hanser.de/Files/Article/ARTK_LPR_9783446415300_0001.pdf`
[9] `file:///Users/ramadanshweiki/Downloads/MOST_Book_2.pdf`

2.4 Keyless Entry

Keyless entry is Mercedes term for an automotive technology which allows a driver to lock and unlock a vehicle without using the corresponding SmartKey buttons[10]. A transponder built within the SmartKey allows the vehicle to identify a driver. An additional safety feature is integrated into the vehicle, making it impossible to lock a SmartKey with keyless entry inside a vehicle. The system works by having a series of LF (low frequency 125 kHz) transmitting antennas both inside and outside the vehicle. The external antennas are located in the door handles. When the vehicle is triggered, either by pulling the handle or touching the handle, an LF signal is transmitted from the antennas to the key[11]. The key becomes activated if it is sufficiently close and it transmits its ID back to the vehicle via RF (Radio frequency > 300 MHz) to a receiver located in the vehicle. If the key has the correct ID, the PASE module unlocks the vehicle. The hardware blocks of a keyless entry ECU are based on its functionality:

- Transmitting low-frequency LF signals via the 125 kHz power amplifier block.
- Receiving radio frequency HF signals (> 300 MHz) from the built-in ISM receiver block.
- Encrypting and decrypting all relevant data signals (security).
- Communicating relevant interface signals with other ECUs.
- Microcontroller.

The keyless entry system also offers other functions than just opening the vehicle door. In this work only opening the vehicle door is considered. Figure 4.3 shows the simplified model of keyless entry use case and the other functionalities explained in this subsection.

Starting the Vehicle

There are two ways of starting the engine on the vehicle: With the Identification (ID) transmitter or the ignition starter switch (start button). If the ignition starter switch is used, it can remain in the ignition lock at all times. To start the engine, it is then only necessary that an authorized ID transmitter is detected in the interior. This type of ignition lock differs from a standard ignition lock in that the key is not turned, but the ID transmitter or ignition starter switch is pushed into the ignition lock.

Switching off the Engine

To switch off the engine, the ignition starter switch/ID transmitter must be pushed completely back into the ignition lock. After releasing the button, it automatically jumps back to the “ignition on” position. The ignition is switched off by further pulling out to the next level. Another difference between the ID transmitter and the ignition starter switch is that the ID transmitter can simply be pulled out of the ignition lock while the ignition starter switch is locked and can only be removed by releasing the unlocking mechanism. To do this, pull the ignition switch to the stop in the ignition lock. Press the release button on the underside of the ignition starter switch and then pull it out.

Locking the Vehicle

Locking like unlocking is possible both actively with the remote control and passively by touching the locking sensor in the door handle. To do this, however, an authorized ID transmitter must be in close proximity outside the vehicle. If the locking sensor is touched once, the vehicle is locked and the ”safe function” is activated. If the locking sensor is

[10] https://www.mbusa.com/en/owners/videos
[11] https://www.hella.com/techworld/de/Technik/Elektrik-Elektronik/Keyless-Go-3195/

touched twice, the vehicle is locked, but the ”safe function” is not activated. ”Comfort locking” is also possible thanks to passive locking. To do this, the locking sensor must be touched for more than two seconds. The system also has a security lock. This means that if the vehicle has been unlocked and a door or the tailgate is not opened within 30 seconds, the vehicle will lock itself again.

Opening and Locking the Trunk

Opening and closing the trunk is possible without unlocking the entire vehicle. If an authorized ID transponder is in the operating range of the rear antenna (see Figure 4.2), the trunk can be opened by operating the tailgate unlocking mechanism. If the trunk is closed again and the authorized ID transponder is in the reception area outside the vehicle, the trunk is automatically locked again.

3. Related Work

This chapter is dedicated to other works that deal with in-vehicle communication networks and the main representatives of publisher/subscriber messaging pattern. Section 3.1 is about the declaration of SOA with regard to in-vehicle communication. Section 3.2 talks about the most important aspects of SOME/IP is used in-vehicle communication network. Section 3.3 of this chapter gives a survey of the currently most frequently used communication protocols in consideration to the publisher/subscriber messaging pattern. In Section 3.4 we compare the protocols selected in the previous section and give an overview of all the features of the protocols.

3.1 Service Oriented Architecture

The software architecture of the Adaptive Platform is based on *service-oriented architecture (SOA)* unlike the signal-based communication paradigm in the AUTOSAR Classic Standard where signals are broadcasted between ECUs [37, 38]. As the Adaptive Platform involves advanced applications like autonomous driving, with Ethernet as the communication medium, it calls for more sophisticated protocols capable of delivering higher payload demands [39].

SOA implements *Service-Oriented Communication (SOC)* where services are exchanged between multiple applications on the communication system. SOC facilitates the scope of new services independent of vendors, products or technologies without much change in the underlying software architecture. As such, communication paths can be established dynamically at run time as required by the Adaptive Platform [40]. The fundamental blocks of SOA and their functionalities are as follows:

Service Providers: register their service in Public Registry.

Consumer Queries: Public Registry for service that match certain criteria.

Service Registry: If Register has such service, it provides consumer with a contract and end-point address for that service.

Figure 3.1: Service-Oriented Communication (SOA) [13].

3.2 SOME/IP

SOME/IP specifies a communication middleware standardized by AUTOSAR for IP-based service-oriented communication in vehicles. It is located on the layers 5 to 7 of the standard ISO/OSI communication layer model, and uses either TCP or UDP as its underlying transport protocol [41]. It aims to include all of the features required by automotive use cases while fulfilling difficult requirements regarding vehicle resource consumption. The problem with classical bus systems (signal-oriented systems) is that the sender decides whether data should be transmitted or not independent of a receivers request. Consequently, a significant amount of non-requested data is occupying the bandwidth which may cause internal communication problems. The SOME/IP middleware is designed to provide a service-oriented abstraction. In contrast to signal-oriented systems, service-oriented systems communicate data exclusively on the receivers demand and not when a sender finds a transmission appropriate, said in another way the main advantage of SOME/IP is that only needed data is send from the server to the client of a subscribed service unlike the traditional way of broadcasting all the data between connected ECUs, hence an increase in qualitative bit rate [42].

3.2.1 SOME/IP Communication Concepts

SOME/IP offers two main communication patterns. First, request/response, implementing classical *Remote Procedure Calls (RPC)* to invoke functions exposed by applications running on remote devices. RPCs are distinguished in *Fire&Forget* and *request-response*. In the first variant, the client calls a method offered by the server, but does not expect a return value (see Figure 3.2). *Fire&Forget* requests are therefore particularly suitable for control commands. The request-response variant is the classic form of a remote procedure call and the calling client receives a response message, which contains a *returnCode* and possible return values (see Figure 3.3). Data fields of a service can be read and changed by clients using *Get* or *Set* methods (see Figure 3.4). In addition, each field can have a *NotifierMethod* which transmits the current value of the field to interested clients.

Second, the *publish/subscribe (PubSub)* approach, which is rather typical in automotive networks. It decouples the sender from the recipients of the messages. A service can offer one or more event groups to which interested clients can subscribe. As soon as the status of the fields belonging to the event group changes in a defined way, the service sends a notification frame (event) with the new status to all subscribing clients (see Figure 3.5). The actual messages are delivered seamlessly by the middleware, which can leverage all the features offered by lower network layers (e.g. multicast groupes) to save transmissions on the communication medium. This mechanism enables signal-oriented communication to be implemented efficiently. The former corresponds to the standard remote procedure call, offering the possibility to invoke functions made available by other applications [13].

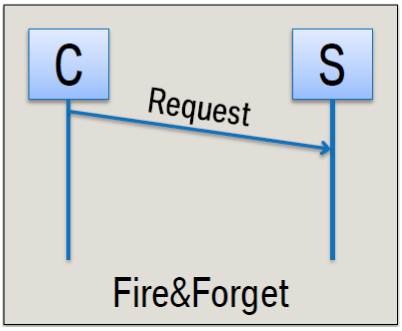

Figure 3.2: „*Fire&Forget*“-RPC: procedure call without answer.

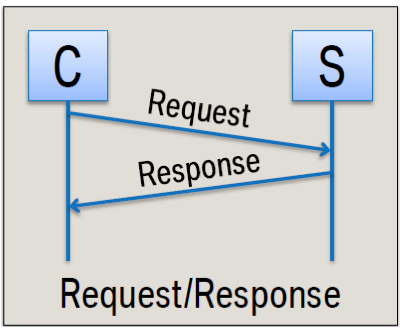

Figure 3.3: *Request-Response*-RPC: procedure call with reply message.

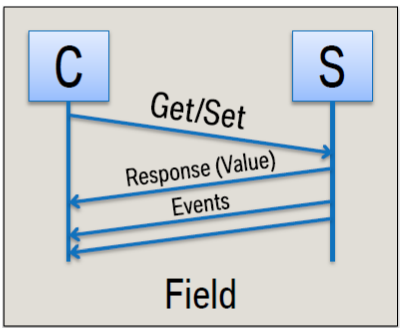

Figure 3.4: Fields: Setting or reading out the data fields another service.

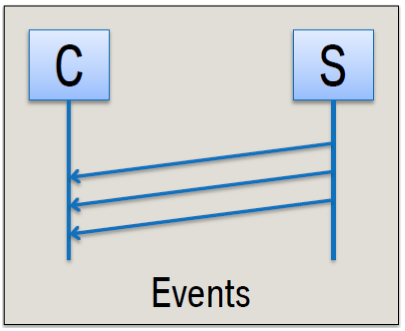

Figure 3.5: PubSub: Clients subscribe to a group offered by the server.

3.2.2 Services

The central element of the SOME/IP protocol, according to the service-oriented paradigm, are the so-called services. A service is defined as a logical combination of methods, data fields and events that are made available to other services. Events describe an abstract change of the service state and allow an efficient implementation of signal-oriented communication similar to the CAN protocol. When such an event is triggered can be defined, for example, by the notifier methods of the corresponding fields. One or more events of a service can be combined into a so-called event group. Clients subscribe to one or more event groups of a service to receive the contained event notifications of the subscribed group. Events can be efficiently implemented with multicast communication should the number of subscribed clients increase.

There are three basic versions of notifier methods supported by SOME/IP. The simplest form *Update-on-Change* sends a notification as soon as the value of the corresponding field changes. If only significant value changes are to trigger an event, so-called *Epsilon-Change* notifiers can be configured. For this purpose, an ϵ-value is defined for the field. As soon as the last value sent changes by more than the defined ϵ, a notify is triggered. As a third variant, a *Cyclic-Update* is possible. For this purpose, the current value of the field is sent at defined time intervals, regardless of a possible change in value.

3.2.3 SOME/IP-Service Discovery

One of the most noteworthy features provided by SOME/IP is the service discovery, which can dynamically advertise the availability of different services as well as manage the subscription to selected events. The SOME/IP *Service Discovery (SD)* functions by services broadcasting so-called offer messages on the network. It is upon receiving such a message that a client, if interested, can subscribe to the service. To reduce subscription time, clients

can also broadcast messages of their own to request a specific service. These are called find messages and to which a client may receive offer messages in response. Another important aspect of the SOME/IP-SD is that a random waiting delay is implemented in the sequence of operations. This ensures that not all services and clients start to operate at the same time, which in turn relieves some network and CPU load. Upon a completed subscription, a subscriber may receive two notification formats: *Event* and *Field Notification*. *Event Notifications* provide data for the specific event and acts as a *Fire & Forget*, whereas *Field* notifications contain data related to previous events and are therefore often equipped with set and get-methods [43].

3.2.4 Systemstart

A single ECU can contain multiple clients and services, there can also be several instances of the same service on different ECUs. The term client refers to a node that is requesting a subscription to a specific service. In SOME/IP terminology, a client can either be active or down and the same goes for a service. After the start of the SOME/IP-SD prozesses, it passes through these three phases: the initial wait phase, the repetition phase and the main phase.

Initial Wait Phase After the system has started, the process enters a waiting phase before the actual SD begins. This waiting period is used to complete the local SD and also to enable slow systems to start and then participate in the SOME/IP-SD protocol. In the best case, all SOME/IP-SD processes in the network are ready and it is sufficient to exchange a single SOME/IP-SD message set per ECU to discover all services in the network.

Repetition Phase After the waiting phase has expired, SOME/IP-SD processes begin to send the composed messages via Internet Protocol (IP) multicast to a preconfigured multicast address. These are made at regular intervals repeated. The interval and total length of this phase is a system parameter and can be selected as required. A typical choice for these parameters is e.g. a total duration of 40 ms with an interval length of 10 ms.

Main Phase In this phase the SD process answers incoming *find service* requests. In addition, a cyclic update of the SOME/IP-SD message can be sent at regular intervals, e.g. to renew the *Time to Live (TTL)* of the services offered [44].

3.2.5 SOME/IP-Transport Protocol

The UDP binding of SOME/IP can only transport SOME/IP messages that fit directly into an IP packet. If larger SOME/IP messages need to be transported over UDP (e.g. of 32 KB) the SOME/IP *Transport Protocol* (SOME/IP-TP) shall be used. The SOME/IP-TP module segments SOME/IP packets that are not suitable for use with a UDP packet and assemble SOME/IP segments received at the receiving end [45]. If it's enabled, the message is split and sent to multiple UDP datagrams. Hard real-time guarantees or reliability mechanisms such as retransmission of lost packets are out of scope here.

Furthermore, all of the prior standards and protocols have been signal-oriented. In contrast, SOME/IP introduces service-oriented transmission of information. SOME/IP specification has thus been an integral part of AUTOSAR since AUTOSAR version 4.1. It was designed to fit for ECUs/devices with limited processing resources and different operation system. SOME/IP supports also the following features needed for the automotive use cases service-based communication approach [46]:

- Small footprint.
- Compatibility with AUTOSAR.
- Scalability for the use on very small to very large platforms.

- Flexibility in respect to different operating systems used in automotive like AUTOSAR, OSEk, QNX, and Linux.
- Relay in using Based on the Transmission Control Protocol/Internet Protocol (TCP/IP) or UDP.

SOME/IP support in-vehicle communication needs such as high data rate, low transportation overhead, and short initiation time. However, two issues remained unsolved:

- The technically suitable solutions weren't useable with AUTOSAR.
- The licensing of the essential patents needed for adapting the solutions were not open.

Although SOME/IP deemed very promising as a communication middleware, but it does not include any security functionality or integrate any security measures, including authentication, integrity and confidentiality leaving the applications and messages transmitted across the network completely unprotected from malicious attacks. Instead, they are delegated to the transport layer, which may impose unwanted limitations and introduce excessive overhead. Because of its service-oriented nature, SOME/IP's network traffic is much more dynamic and unpredictable compared to classic automotive protocols like CAN. As a result, the capability to monitor the system and, in particular, the network traffic between its individual components is crucial for ensuring an adequate level of security and preventing large-scale attacks targeting entire vehicle fleets. In addition, there are no application firewalls for SOME/IP that would extend packet filtering to some basic anomaly detection [11, 47, 48]. For technical details about SOME/IP we refer to Chapter 2.

3.3 Publish/Subscribe Protocols

In the IoT environment, there are various communication patterns. Request–response or request–reply provides that a client sends a request for some data and a server responds to the request. In contrast to request–response, publish/subscribe is not based on requests and responses, but on publishing and subscribing to information. In the publish/subscribe model, subscribers typically receive only a subset of the total messages published. There are two common forms of selecting messages for reception and processing called *filtering*: topic-based and content-based. In a topic-based system the publisher specifies the topic string when it publishes the information, and the subscriber specifies the topic strings on which it wants to receive publications. The subscriber is sent information about only those topic strings to which it subscribes. In a content-based system, messages are only delivered to a subscriber if the attributes or content of those messages matches constraints defined by the subscriber. The subscriber is responsible for classifying the messages. This section describes the most important publisher/subscriber protocols. These are the following: the MQTT, AMQP, CoAP, XMPP, ROS, ZeroMQ.

3.3.1 Message Queuing Telemetry Transport (MQTT)

It is a *machine-to-machine (M2M)/ Internet of things (IoT)* connectivity protocol that was designed as an extremely lightweight publish/subscribe messaging transport. It was originally developed by IBM and is now an open standard. First introduced in 1999 and adopted nowadays [49]. Starting from version 3.1, MQTT has become an OASIS standard and the latest release of the specifications refers to version 3.1.1 of the protocol (OASIS, 2014) [50]. The MQTT protocol is based on TCP/IP. A variant *MQTT for Sensor Networks (MQTT-SN)* uses UDP as its transport protocol [51]. MQTT uses the broker-based publish/subscribe pattern for exchanging messages. A client who wants to get the massage belong to certain topic subscribes topic to a message broker server. If another

client publishes the message with the certain topic, then a message broker server re-publish the message to the subscriber who already subscribes with the certain topic. Topic is a message string to filter messages for each client. Foster et al. [52] states in his paper that MQTT is not capable of fulfilling hard real-time requirements. MQTT supports some basic QoS to define if and how often a message should be re-sent until it is acknowledged by the broker and if the server should cache topic data. There are no security mechanisms supported by MQTT. But since MQTT version 3.1, there is the possibility of transmitting a user name and password in a MQTT message in order to facilitate the authentication of individual clients.

3.3.2 Advanced Message Queuing Protocol (AMQP)

It is a lightweight M2M protocol, which was developed by John O'Hara at JPMorgan Chase in London. It's first version 1.0 introduced in October 2011 standardized by the OASIS in 2012 [53]. It uses a reliable transport protocol TCP or SCTP for transporting data. AMQP is a communication protocol that is capable of supporting both, the request/response and the broker-based publish/subscribe pattern. As such, this protocol is not suitable for real-time communication [51]. Reliability is one of the core features of AMQP, and it offers three preliminary levels of QoS or delivery of messages. The AMQP provides complementary security mechanisms, for data protection by using TLS protocol for encryption, and for authentication by using SASL [52].

3.3.3 Constrained Application Protocol (CoAP)

It has been designed to offer simplicity, low overhead and machine-to-machine communications that are required to enable the interaction and management of embedded devices. It was developed by IETF *Constrained RESTful Environments (CoRE)* working group, first introduced in 2010 and standardized in 2014 [51]. CoAP uses UDP as a transport protocol and supports request-response and publish/subscribe architecture [54]. This protocol is not recommended for fast communication. Foster et al. [52] considered CoAP as not real-time capable. Despite using UDP, CoAP has two different levels of QoS, i.e. *confirmable* and *non-confirmable*. Messages marked as *non-confirmable* do not have to be acknowledged while *confirmable* messages require an acknowledgement by the receiver [55]. As security is important to protect the communication between devices *Datagram Transport Layer Security (DTLS)* is used. Apart from DTLS, sometimes IPSec is also considered.

3.3.4 Data Distribution Service (DDS)

It was developed by the Object Management Group (OMG) for a M2M communication und first released in 2004. It is an emerging specification for publish/subscribe data distribution systems. The purpose of the specification is to provide a common application level interface that clearly defines the data distribution service. It uses UDP as its transport protocol by default and TCP if necessary. DDS supports the brokerless publish/subscribe pattern [56]. This protocol was designed to address the needs of time critical applications and it is therefore capable for real time communication. DDS provides a rich set of QoS, with which users can configure and control the local and end-to-end properties of DDS entities to meet the application requirements. The so-called QoS policies can be used to address the needs and requirements to a message, such as reliability, resource utilization and latency of the data [52].

3.3.5 Extensible Messaging and Presence Protocol (XMPP)

It is an open standard messaging protocol formalized by IETF and developed for instant messaging applications. It uses TCP as its transport protocol. XMPP client/server

architecture supports the publish/subscribe pattern, as well as the request/response pattern [57]. It hs been specified as a protocol that is capable of near real-time communication. Near real time describes the delay introduced, by automated data processing, between the occurence of an event and the use of the processed data [58]. In terms of security, there are two security mechanisms defined, which include the use of *Simple Authentication and Security Layer (SASL)* for secure authentication as well as the use of *Transport Layer Security (TLS)* for achieving encryption and confidentiality.

3.3.6 Robot Operating System (ROS)

ROS is an open-source robotics middleware that provides low-level device control, package management and messages passing between processes. It connects processes of programs, known as nodes, that perform different functions [59]. Initial release on 2007 developed by Willow Garage and supported by *Open Source Robotics Foundation (OSRF).* ROS currently supports TCP/IP-based and UDP-based message transport. (TCPROS) is the default transport used in ROS and is the only transport that client libraries are required to support. The UDP-based transport, which is known as (UDPROS) and is currently only supported in *roscpp*, separates messages into UDP packets. It is not real-time, because it is built on top of Linux, which is not inherently real-time [60]. RPC is natively supported by ROS. ROS only supports authentication via MAC-Address using third-party packages.

3.3.7 Zero Message Queue (ZeroMQ)

ZeroMQ (known also as 0MQ or zmq) as the name indicates, it is a messaging queue but unlike regular MOM, it is a messaging library, which allows asynchronous communication of some devices through the network protocol. The zero in the term refers to zero broker (ZeroMQ is brokerless) and therefore is the message queue unlike MOM can run without a dedicated message broker. The development of ZeroMQ by iMatix corporation started in 2007 with the aim of developing a lightweight and easily scalable distributed messaging system [61]. ZeroMQ works in request-reply pattern as well as the publish/subscribe pattern using TCP type sockets connections. To establish a middleware communication, all participants have to know each other, therefore, ZeroMQ is a messaging framework and not a fully MOM [62].

3.4 Summary

Table 3.1 gives an overview of the basic features for each of the presented middleware protocols in the previous section that support PubSub communication mode. The first column lists the most important MOM protocols. The order in which the protocols are entered does not reflect the importance of the protocol. The second column shows which protocols support the request-response (marked with ✓) and doesn't support (marked with ✖) this communication pattern. The third column shows whether the protocol is standardized and licensed and by whom. The fourth and fifth column show the transport protocol used by default and for certain types of communication and whether this communication is possible in real-time. Among the protocols in the list (in the sixth column), there are also some protocols that support different levels of QoS. The seventh column with the name Broker-required indicates whether a broker is mandatory for the functioning of MOM protocols. No Broker-required can also mean that a broker is optionally possible in this protocol. In case of security mechanisms (to see the eighth column) some of the protocols support one or more security mechanisms or none at all marked with '✖'. The last column is about which protocol supports semantic data. This means a model describes the meaning of its instances.

The evaluated *Message Oriented Middleware (MOM)* protocols have different feature listed in Table 3.1. Request-response pattern can be implemented in the most listed protocols and is an important and often-used feature. Most of the interactions in distributed systems are based on the request-response messaging paradigm, where a client uses a channel to send a request to a server that, in turn, sends back a response. In some cases it's the most frequent solution to an integration requirement and the best option. MQTT and DDS doesn't support request/response model. There uses data-centric paradigm and it's more efficiently realized by a publish/subscribe communication model. MQTT, AMQP and CoAP are not designed for hard real-time communications. There are some MOM protocols that support QoS and some that do not since it plays a critical role in the overall system performance. Protocols like MQTT, AMQP, CoAP and XMPP only work with one broker other than the rest they can work with or without. ZeroMQ and ROS support real-time communication but doesn't include any security mechanisms. OPC UA has its strength in the semantic modeling of information. OPC UA is currently the only publish/subscribe protocol in the Industry 4.0 and IoT environment that supports a semantic model, which we found out through our research. OPC UA semantic model called (information model) provides semantic information which gives the data context and meaning, so any human or device can understand the data correctly. For autonomous systems that is one of crucial features, since it provides better interface for machines or entire system units to query information of other machines or system units to find out their services, interfaces and capabilities. The OPC UA information model enables more efficient and autonomous interaction between systems. Furthermore, OPC UA supports security mechanisms, which include authentication, authorization, confidentiality and integrity. Each of these protocols has their different pros & cons. They are designed for particular scenarios. So it is very difficult to prescribe any single protocol for diverse IoT applications.

For this work we have searched for a protocol that supports publish/subscribe and is also brokerless functional. Furthermore, it is also important in this thesis to work with protocol that supports the semantic model and from the Table 3.1 we can see that only OPC UA support this feature. This feature makes OPC UA a device-centric protocol that focuses on device interoperability, where devices may be used in different systems. In addition, OPC UA include the possibility to add additional information for every transmitted data package which enables the transmission different kind of data in the same connection.

A clear answer to the research question **R2.3** is that OPC UA doesn't create a dedicated TCP connection for every subscriber and publisher pair as e.g. MQTT. OPC UA need to include additional information about the published data in the transmitted package. The answer to research question **R2.4** is that OPC UA PubSub does not include any QoS mechanisms itself as can also be seen in the Table 3.1. But in combination with, e.g. TSN on layer 2, OPC UA PubSub can also support additional QoS principles.

Protocol	Req.-Rep.	Standard	Transport	Real-time	QoS	Broker-required	Security	Semantic Data
MQTT	✖	OASIS	TCP	✖	3 Levels	✓	TLS/SSL	✖
AMQP	✓	OASIS	TCP	✖	3 Levels	✓	TLS/SSL	✖
CoAP	✓	IETF	UDP	✖	Limited	✓	DTLS	✖
DDS	✖	OMG	TCP/UDP	✓	Extensive	✖	TLS/DTLS/DDSsec	✖
XMPP	✓	IETF	TCP	✓	✖	✓	TLS/SSL	✖
ROS	✓	OSRF	TCP/UDP	✓	✖	✖	✖	✖
ZeroMQ	✓	GNU GPL	TCP, IPC	✓	✖	✖	✖	✖
OPC UA	✓	OPC UA	TCP/UDP	✓	✖	✖	PKI, TLS, ACL	✓

Table 3.1: Comparison of protocols main features supporting PubSub.

4. Approach

This thesis investigates the applicability of an Industry 4.0 protocol for in-vehicle communication by modeling a complex communication scenario in OPC UA and developing a prototype application for further evaluation. For this we identified the keyless entry of a vehicle, which means open the vehicle door without holding the key in the hand. Section 4.1 describes the model of the system, including a detailed description of all involved components and their interactions with each other. In Section 4.2 an information model is described that defines the overall structure of the use case, what kind of information is exchanged and how the exchanged data is represented. Section 4.3 explains how applications benefit from the keyless entry information model. Lastly, Section 4.4 deals with the concept of publish/subscribe application, including the requirements for our prototype and the implementation of this prototype.

4.1 System Model

The keyless entry system no longer requires active use of the key by pressing the remote control or even operating the door lock manually. It is sufficient for the driver to carry the radio key with him, e.g., in his trouser pocket. There is no need to search for the radio key, it no longer has to be held in the hand to actively trigger the desired functions by pressing the radio key buttons. These access and drive authorization systems increase convenience and simplify access to the vehicle. A variety of system components must be added to the traditional central locking system in order for the passive keyless entry function to operate. In this section we first describe the required component for keyless entry system then we describe the procedure of keyless entry and how can this concept be mapped to a OPC UA *publish/subscribe (PubSub)* communication.

4.1.1 Involved Components in Keyless Entry

Keyless entry provides several functionalities. Unlocking the vehicle is one of the most popular and recently used. This work only considers unlocking the vehicle, as this is the most complex functionality of keyless entry. This section describes the components of a keyless entry system that are commonly built in nowadays. We simulate a functionality of the keyless entry system, namely the unlocking of the vehicle door. The keyless entry system offers other functionalities than just opening the vehicle door like locking the vehicle, start/stop the vehicle engine and opening and locking the trunk explained in detail in Chapter 2. Figure 4.3 shows the simplified model of keyless entry use case and

other components and functionalities. Figure 4.1 shows approximately where the required components are located in the vehicle and how they are connected to each other. The involved parts of the keyless entry system required to open a door are described in the following [63]:

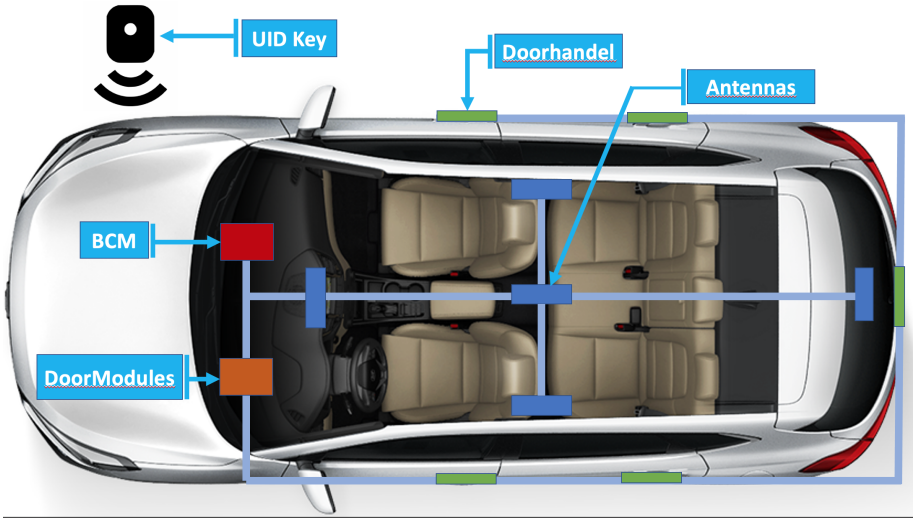

Figure 4.1: Required components of keyless entry in vehicle.

Door Handle

The vehicle door handles contain receiving antennas and locking sensors for detecting when the vehicle should be opened or closed. There are only sensors in the rear door handles to detect whether the vehicle should be open or closed. The proximity sensors work according to the principle of capacitive sensors. As soon as the driver's hand is within the range of the proximity sensor, the capacitive sensors sense and relay this in the form of a signal to the comfort control device.

Kessy I/O

The *KessyI/O* is a compound of multiple antennas, which can be seen in Figure 4.1. It contains antennas in the interior of the vehicle and others on the outside. This include the rear antenna on the outside, which is built into the rear bumper and is in charge of reception at the rear of the vehicle. The interior contains the antennas for the interior, trunk, and parcel shelf. The inner, trunk, and rear antennas each consist of a condenser ferrite coil and are constructed as a series of resonant circuits. A versatile *Printed Circuit Board (PCB)* with a conductive loop that serves as the field generator is the parcel-shelf antenna.

ID Transponder

It is a radio key and consists of a simple remote control and an identification transmitter. Active unlocking and locking over distances of up to 100 m from the vehicle requires manual key operation on the ID transponder. Passive unlocking is achieved by engaging the door handle, while passive locking is achieved by touching a sensor surface on the door handle. For safety reasons, passive operation is only possible in the immediate nearness of the ID transmitter to the vehicle (approx. 2 m). The ID transponder communicates with the control unit by radio. It also includes a mechanical emergency key with which the driver's door can be unlocked manually, for example when the radio key battery is empty.

Body Control Module

Body control module (BCM) is a very complex unit which is responsible for many functions. It is used, for example, to monitor external and internal lighting, windows, seat heating, car alarm, door opening (door and trunk lock), velocity and odometer, rear window and mirror heating, and so on. The BCM is a multifunction module that operates the keyless entry system. When an radio frequency message is received from a keyless entry transmitter, the BCM interprets this signal and performs the specific function, i.e. door lock, door unlock, or vehicle locate.

Door Modules

Door modules need to be able to control standard loads efficiently, in their simplest form, such as those presented by door locking motors. The chipset needs to monitor not only several standard loads such as door lock motors, mirror directories and levellers in more complex door zone systems, but also those for defrosters and many lighting functions, from LEDs to incandescent bulbs.

There are also other components and aspects for our use case (unlocking the door vehicle) but are not considered as a single component and implemented together with the previous components:

Control Unit

It controls a number of functions, the central locking system, the comfort opening/closing of the side windows and sunroof and other functions. It is also the responsibility of this control unit to detect errors and store them in the system. Communication between a radio transmitter key and the control unit, depending on the area.

Radio Communication

Messages are exchanged between keyless entry components to check if the user has an authorised ID transmitter in order to perform a function on the vehicle. The control unit sends a signal, triggered by the capacitive proximity switch on the door handle, to the ID transponder and receives a response from it. On the signal sent by the vehicle via the Low Frequency (LF) antennas, not only can data be transmitted to the ID transponder, it also serves to uniquely locate the ID transponder. In this way, it can be determined with high precision whether it is inside or outside the vehicle. This position determination must be very precise so that the control unit can ensure whether an authorized ID transponder is inside the vehicle and start authorization can be granted or whether an ID transponder located inside the vehicle must be deactivated after locking if the vehicle is locked from the outside by a second, authorized ID transponder.

Radio-Frequency Identification (RFID) Zones

Two RFID zones are required, as depicted in Figure 4.2, with one reaching outside of the vehicle (generally in a range no more then one meter from the doors) for unlocking and one exclusively on the inside of the vehicle for starting the engine. In this thesis we consider the opening of the vehicle door. While the LF zones are strictly in a few meter radius around the car the range of the ID transmitter can be significantly larger [64]. The RFID zones of keyless entry system are made with the antennas. In addition to the antennas in the door handles, other antennas are integrated in the exterior and interior of the vehicle. In the outdoor area, this includes the rear antenna. In the interior are the interior, trunk and rear shelf antennas. The radio connection is established with the help of the antennas in the door handles and the ID transponder.

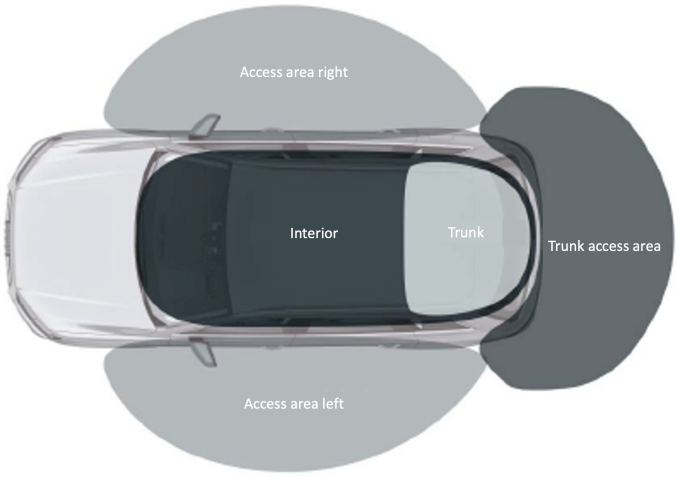

Figure 4.2: Inner and outer RFID zones [65].

4.1.2 Procedure of Keyless Entry

We identified all the components required for our keyless entry use case. Now we will describe how keyless entry system works with the requested components. With the keyless entry system, a driver is offered an automated unlocking system to open the vehicle door. The user of the keyless entry system can move towards his vehicle while it is unlocked fully automatically, as soon as the key is in a predefined area of the vehicle and the user touches the *DoorHandle* (as seen in Figure 4.2) the vehicle door opens. The comfort key system determines the position of the ID transmitter (vehicle key) in relation to the vehicle. Functional prerequisite for opening, the key is within the defined range in the vicinity of the vehicle. In addition, a check is made whether the ID transmitter is authorized for this vehicle by checking an internal electronic/logical key for validity. If the ID transponder is recognized as authorized by the control unit, the central locking system is activated and the vehicle is opened. Depending on the coding, individual or total locking, it is then possible to open the doors. In various works, it is stated that keyless entry system needs approx. 50 - 60 ms time to open the vehicle door [63]. The process, from activation of the sensors in the door handle to unlocking the vehicle. For locking the vehicle door, it is sufficient if the *User Identifikation (UID)* key is outside the defined RFID zones range formed by antennas.

Figure 4.4 shows the complete communication between all entities of the use case. On the left side (y-axis) the time unit is indicated, when each message is sent and how long it takes for the message to arrive. At the top (on the x-axis), the entities are listed according to the meaningful order of the communication process.

When a user approaches the vehicle and pulls or touches the vehicle door handle, the *DoorHandle* is triggered and sends as first a *Hardwired* (HW) trigger/wake-up message to the *KessyIO*. The data type of the sent message is Boolean. The vehicle user can unlock any vehicle door with keyless entry system even the trunk door.

KessyI/O receives the message from the *Doorhandle* and the component is woken up. Directly after the component has been woken up it sends a (HW) trigger/ *WakeUpRequest* to BCM. Then it sends another message to UID (key) it takes more than 25 ms for the message to arrive. It is a *Low Frequency (LF)*Data message with a UInt32 as data type. It sends the data via LF antennas to UID (key) to clearly locate it. This enables the *KessyIO* to differentiate precisely between the interior and exterior of the vehicle. After about 60

ms a CAN entry validation message is sent to BCM. The sent LFData of data type Integer come from the sensor on the door handle for detecting where and when the vehicle should be opened or closed. This is detected by the capacitive sensors and transmitted to the comfort control unit (BCM) in the form of a LF signal. The key reacts to this and sends an *High Frequency (HF)*Data with the data type Double to BCM. Before the message arrives, BCM is awake and ready for operation.

The UID key, which is in the driver's hand or pocket, receives an LFData message from *KessyIO* (the vehicle's antennas). The key reacts to this and sends an HFData with the data type Double to BCM. Before the message arrives, BCM is awake and ready for operation. It is response message to the control unit for checking the authorization. A *WakeUpRequest* is also sent to the *DoorModules* immediately after the BCM wakes up. The vehicle *BCM* verifies this reply and transmits an frame consisting of the vehicle identifier (a challenge and a signature). In turn, the UID key uses this challenge in the compression function to produce a response before transmitting it over to BCM. The BCM checks whether the ID transmitter is authorised for this vehicle by checking the validity of an internal electronic/logic key. If the control unit recognises the UID key as being authorised, the central locking system is activated and BCM sends as last a message to the *DoorModules* to unlock the vehicle door. Depending on the encoding individual or all doors can then be opened.

Figure 4.3 shows a simplified model of keyless entry use case. The other functions and components of the keyless entry use case are also entered in the model. Not all messages that are exchanged are shown as in the sequence diagram in Figure 4.4. *DoorModules* and UID key communicate with the BCM (here *KessyIO* is shown as a one component). Messages are exchanged between the components with different contents. If it is the correct key and the key is in the correct position, the BCM component sends an unlock message to the *DoorModules* via CAN network to open the vehicle door. This section answers all research questions under **Goal 3** and successfully fulfills it. We chosen a complex use case in-vehicle communication (the keyless entry). We then explained the use case in detail with all its components and the messages that are exchanged between them.

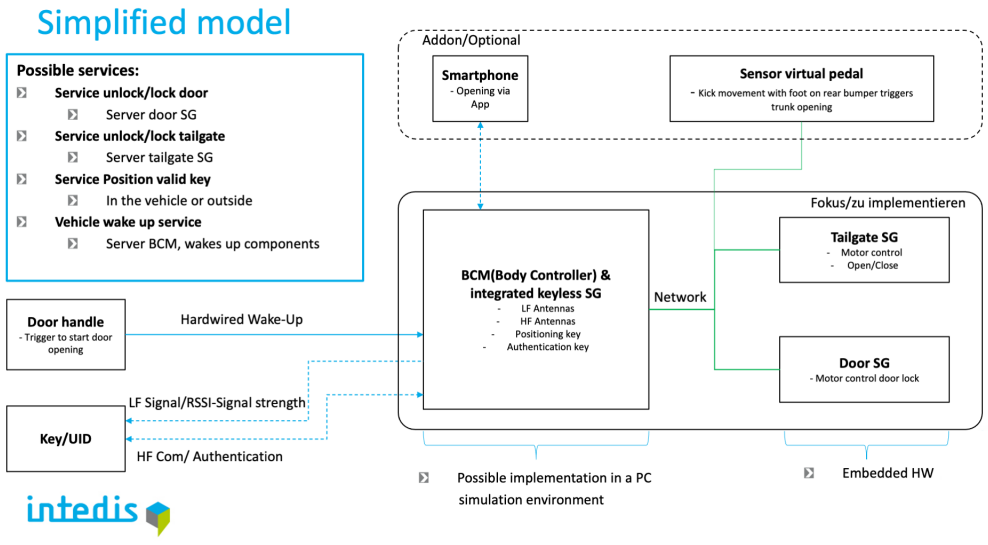

Figure 4.3: Keyless entry simplified model [65].

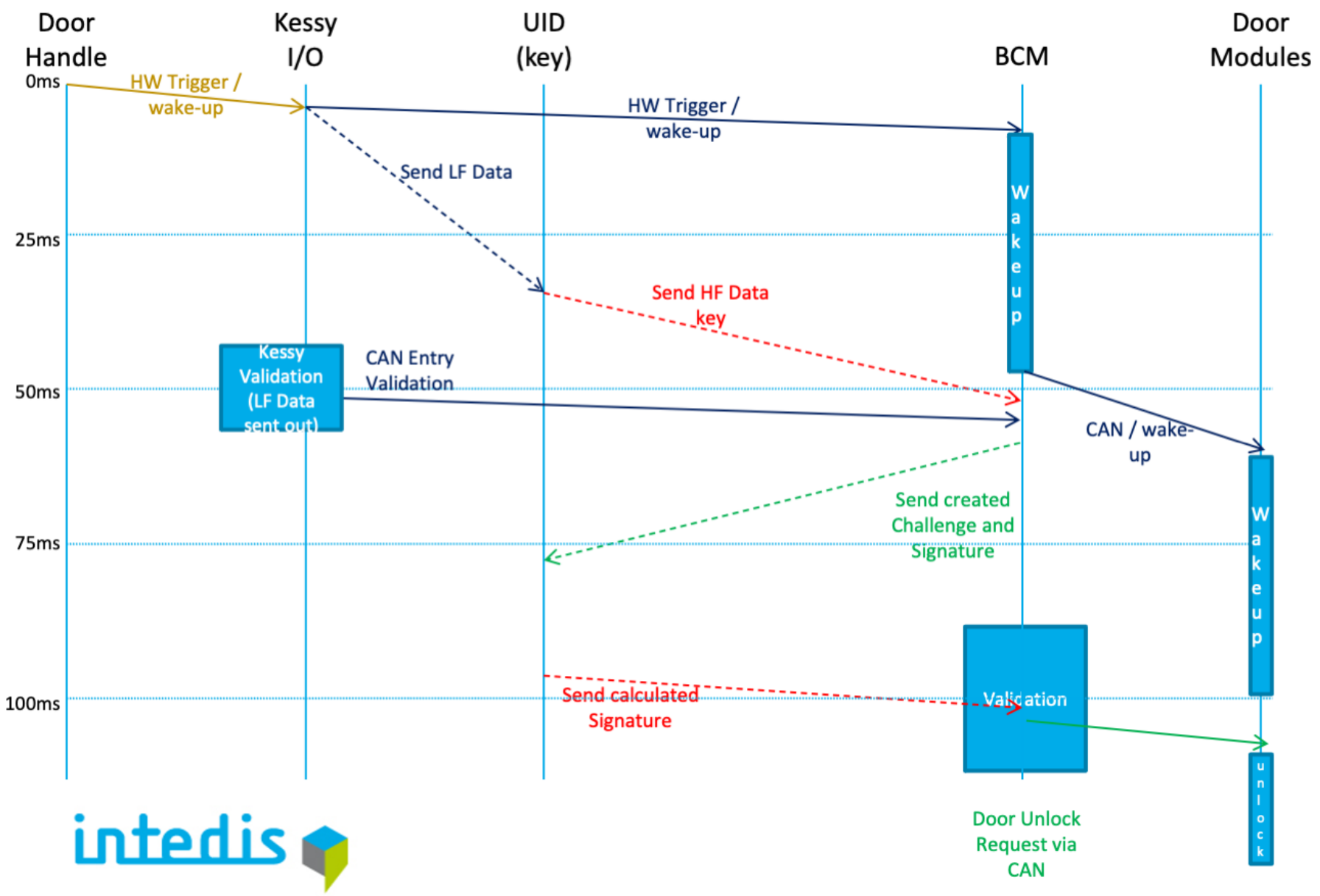

Figure 4.4: Keyless entry timing chart [65].

4.1.3 Concept of Keyless Entry with PubSub

We described in the last section how keyless entry normally works. Keyless entry components normally communicate directly with each other. This section describes our concept for a publish/subscribe-based keyless entry and how it works. Each component in keyless entry publishes or subscribes or both. A publisher can create one or more topic messages, each topic message has different contents. A subscriber subscribes to one or more publishers and subscribes to a one or more topic from the same publisher.

Table 4.1 gives an overview of the keyless use case implementation with PubSub. In the first column, all topics that were exchanged between the components in the entire use case are entered. The topic messages are listed according to the order in which the use case was implemented. The second column shows which component created the topic and published it to the subscriber. The third column shows which component has subscribed the topic message.

Topic Name	**Publisher**	**Subscriber**
HW Trigger/ Wake up request	DoorHandle	KessyI/O
HW Trigger/ Wake up request	KessyI/O	BCM
Send LFData	KessyI/O	UID (key)
Send HFData	UID (key)	BCM
CAN / Wake up request	BCM	DoorModules
CAN Entry Validation	KessyI/O	BCM
Send created Challenge and Signature	BCM	DoorModules
Send calculated signature	UID (key)	BCM
Door unlock request	BCM	DoorModules

Table 4.1: Shows CPU and memory utilization of the keyless entry components.

In addition, it has become clearer which components play which role. Table 4.1 shows that the *DoorHandle* is a pure publisher component that only publishes a message once when the *DoorHandle* is pulled or touched. *KessyIO* only subscribes to *DoorHandle* when the component is woken up, then it only publishes and not subscribe again until the component switches to sleep mode. It publishes to BCM a wake-up request and CAN entry validation and publishes HFData to UID key. The UID key is a publisher and subscriber component. When the key is in the defined range, it subscribes to an LFData message from *KessyIO* antennas. The UID key publishes a HFData back to BCM and now subscribes from BCM a signature message, which it publishes back again.

BCM is a publisher subscriber component. After the component first subscribes to a topic from *KessyIO*, it directly publishes a CAN wake-up request to the *DoorModules*. It also subscribes a HFData key from UID key and a CAN Entry validation message from *KessyIO*. It publishes created signature to the UID key and subscribes the result. At the end, the message is validated and a door unlock request message is published to *DoorModules*. The last component is *DoorModules*, which is responsible for opening and closing the vehicle door. *DoorModules* is a pure subscriber component. It subscribes only the topic of BCM like a CAN wake-up entry request and the last message is door unlock request, so that the vehicle door is opened at the end of *DoorModules*. In this section the **Goal 3** and four research questions have been answered. We chosen the keyless entry use case and explained all its components and their messages that are exchanged between each other.

4.2 Information Model

OPC UA information models are representations of data that specify how data of different domains is presented in an OPC UA address space. Information model is usually an abstract, formal representation of entity types that may include their properties, relationships and operations. It provides formalism to the description of a problem domain without constraining how that description is mapped to an actual implementation in software [66]. This is important to ensure a good definition and understanding of the complex systems. This section describes the information model, that defines in detail, which information related to the keyless entry use case in OPC UA is exchanged in a well-structured and standardized manner. The structure is defined in an information model.

Accompanying the presentation of our developed structure for the keyless entry use case, we explain in this section how such a model is created. It is created according to the following graph in Figure 4.5. It starts after the *setup open62541* explained in earlier section, since the library was installed and used before. What is not considered are the last two steps *connect application* and *client validation*, because in this work PubSub is used and not client-server. An information model defines the nodes and their structure provided in the server's address space. Its semantic is one of OPC UA's strengths. An information model, similar to object-oriented programming, describes types that can be expanded and instantiated. Furthermore, by using specific reference types for other nodes, these types can be semantically enriched. A set of nodes and their references is an OPC UA information model [67].

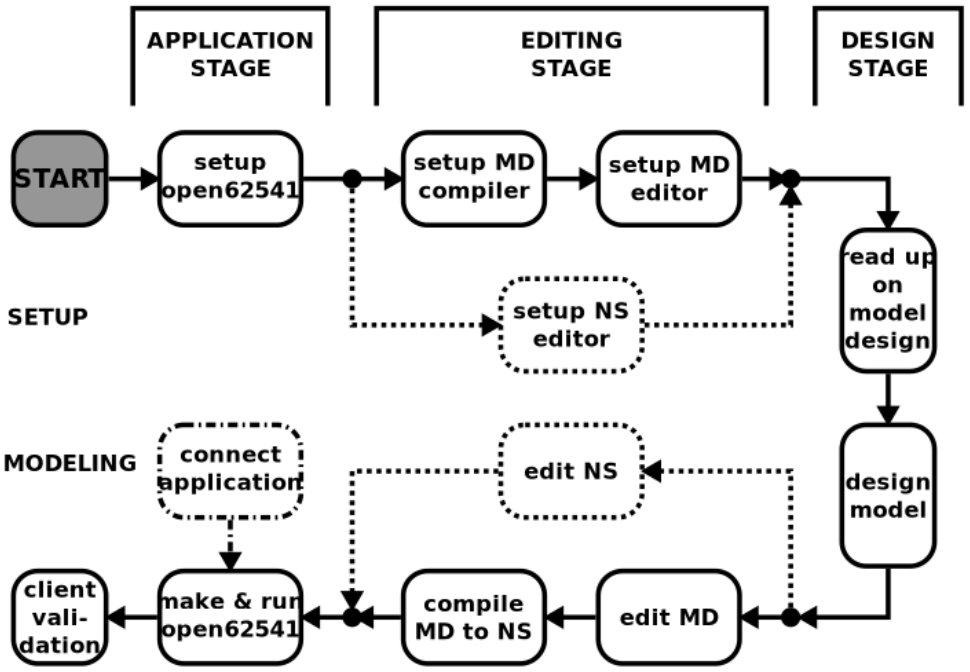

Figure 4.5: All steps required to generate a custom UANodeSet [68].

4.2.1 Abstract Representation of Communication

We described in Section 4.1 all involved components of keyless entry use case with the exact description of all its components individually and how they all interact with each other. According to the commonly communication procedure described in Section 4.1, the

DoorHandle ist the first required component. When a driver approaches the vehicle with the vehicle key and pulls or touches the *DoorHandle*, a message is sent to *KessyIO* (antennas or receivers). It is a *WakeUp* message via receiving antennas, with the *DoorHandle* directly connected to the *KessyIO* system. The *KessyIO* wakes up and is now ready to inform the other components that a door opening request has been sent.

The *KessyIO* has a direct connection to the Body control module (BCM) whereby a *WakeUp* message is sent to BCM. The *KessyIO* sends a LFData to the UID key. The UID key receives the message when it is in the predefined area around the vehicle. The BCM later receives another confirmation that an LFData is sent to the key. The BCM has woken up to a request from *KessyIO* and is now ready to verify that it is the correct key. The UID first sends an HFData to BCM. Then a signature and challenge message is sent from BCM to UID to verify that it is the correct key for the vehicle. After that, BCM receives from UID keys the calculated signature for the validation of the keys. BCM sends a *WakeUp* message to *DoorModules*. *DoorModules* is woken up and ready to unlock the vehicle door, after BCM checks the sent data and sends a request *UnlockDoor* via CAN.

We now need to model the application structure and information exchange for keyless entry. We have created an information model that defines what are the data types of the exchanged information and how can it be structured. Before we model the appropriate structure and representation of the exchanged data, we first need to explain a graphical notation for modelling an OPC UA information model for our keyless entry use case and the steps we took to create an information model.

4.2.2 Steps to the Information Model

This Section describes the steps for creating an information model for the implemented keyless entry use case. Generally speaking information models are sets of readymade types of objects, variables, references, events and data. OPC UA defines a graphical notation for modelling an OPC UA address space of the application. Chapter 2 gives an explanation of the most important predefined *ReferenceTypes*, *ObjectTypes* and *VariableTypes* which are used to create a graph based information model. Create and import OPC UA information model for the implemented keyless entry use case and for other use cases goes through the following steps in Figure 4.6. Prerequisite is a basic knowledge of OPC UA information models[1]. The previous section does not explain all available structure of information models concepts provided by the OPC UA specifications, but introduced the most important data model.

First comes the actual idea of how the application is uniformly structured. How is the application structured, how does the application work and how is it defined. Which components are there, which variables, methods and relationships are there, what exactly is exchanged between the components. We have created a metamodel that contains ideas and concepts of our use case. It is written in an XML language. There are some examples of metamodel for the creation of an information model from other use cases [2] [3][4].

[1] https://opcfoundation.org/developer-tools/specifications-opc-ua-information-models
[2] https://github.com/Pro/opcua-animal-cs
[3] https://github.com/OPCFoundation/UA-Nodeset/tree/v1.04/PLCopen
[4] https://github.com/OPCFoundation/UA-Nodeset/tree/v1.04/MTConnect

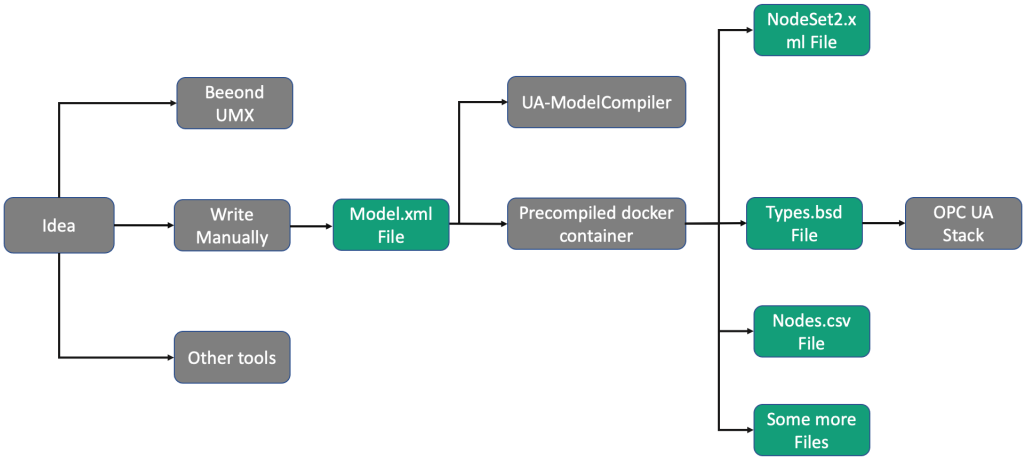

Figure 4.6: Steps to create custom information model.

Beeond UMX Pro Tool, it allows an easier GUI creating and editing of information models. It is not necessary to write the metamodel itself from scratch, but the understanding of information model notation as explained in detail below is sufficient here with the GUI tool that drag and drop the components creates a diagram and then the diagram can be converted automatically into metamodel. In this thesis, we have written our ideas and preliminary designs for the structure of the keyless entry use case in a text editor. To better detect errors the script was executed from time to time. And so we have the metamodel for our use case explained in details in Section 4.2.3 ready for compilation.

The second step is to compile the created metamodel contains the ideas and considerations for the structure of the use case with a specified compiler *UA-ModelCompiler* of OPC UA foundation. There are two ways to compile it with the *UA-ModelCompiler*:

- Linux: Using a precompiled docker container (recomanded).
- Linux/Windows: Using the UA-ModelCompiler sources and compilation the binary. It's based on *.NET*.

To compile our metamodel written in an XML file called *KeylessEntry.xml* we used precompiled docker container. On Linux it is strongly recommended to use the precompiled docker container available on DockerHub[5]. The following instruction in Listing 4.1 is to run the precompiled docker container. The UA-ModelCompiler compiles or converts the metamodel into another form consists only a set of nodes like XML schema called *NodeSet*. This contains the content of the information model. The *UA-ModelCompiler* outputs the following several files [69]:

- .csv: Used to generate the NodeID header to provide defines for specific node IDs instead of numbers.
- .bsd: Used to generate custom data types and struct definitions if you define your own data types in the nodeset.

[5]https://hub.docker.com/r/sailavid/ua-modelcompiler

- NodeSet2.xml: Contains the whole nodeset definition and is used to generate the initialization code for the open62541 server.

Listing 4.1: Compiling metamodel to create an information model with docker.

```
1 docker run \
2     --mount type=bind,source=$(pwd),target=/model/src \
3     --entrypoint "/app/PublishModel.sh" \
4     sailavid/ua-modelcompiler:opcua_rocks_tested
5      \model\src\keylessEntry kessy /model/src/Published
```

The *NodeSet* created from the compiled metamodel is contained in *NodeSet2.xml* and it's the information model we have used in this work. The information model contains a set of nodes that are mapped by objects, variables and methods that we created in the metamodel. With other words the generated *NodeSet* is a graph-based data structure that contains the content of information model, in which typed nodes are linked through typed references. Keyless entry information model is described in details in Subsection 4.2.4. The next step is to use the generated *NodeSet* in any supported stack or modeler. Open62541 stack supports code generation from the information model. Furthermore, we have created graphical notation for the use case from the information model see Figure 4.12.

It's possible to write the information model (*NodeSet*) manually. This would be really cumbersome and may lead to invalid and inconsistent set of nodes. Therefore this is not recommended. Another possible solution would be to just use the provided server API and create all nodes through the corresponding API calls. This is not a portable solution and should also be avoided. In Chapter 2 shows the most common open-source and commercial tools offered by OPC UA for creating information models, their use is not necessary.

4.2.3 Metamodel for Information Model Implementation

After we have explained the keyless entry components in detail in Section 4.1, and we know exactly how the communication between the components is built. Who communicates with whom and what kind of messages are exchanged. It is also important to understand the information model notation as a whole picture of what will later be generated as an information model from the application. This metamodel is written in an XML language. These ideas and preliminary considerations are intended as a first step towards creating an metadmodel for the information model implementation file from the keyless entry application. We have presented in Section 4.2.2 important steps and tools that are helpful to create metamodel of the application like using text editors or other advanced tools from the OPC UA foundation and from other manufacturers.

At the beginning the namespace Uniform Resource Identifier (URI) have to be defined. Every information model must have its own unique URI. In the next step we extend the metamodel with *ObjectTypes* which objects can be instantiated later in the model.

First we defined an *ObjectType KessyType* for the application as an abstract class for all other keyless entry components. It obliges all subclasses the Id of the class representing a keyless entry component and the URI (IP address and the port number). *Doorhandle* is the first component of keyless entry introduced according to the communication sequence diagram in Figure 4.4. *Doorhandle* is the first component that triggers the communication when the *Doorhandle* is activated. We created an *ObjectType* called *DoorHandleType*. A *Doorhandle* can be objectively instantiated using this type. Each *Doorhandle* object must have a unique Id, IP address and port number.

We have defined Ids for all keyless entry components as UInt32 as suggested in the open62541 PubSub and IP address with the port number as a variable of type String. *DoorHandle* publishes a *WakeUp* message, which is subscribed by *KessyI/O*. We have created a *VariableType WakeUpRequestType*, which creates a variable of type Boolean. This variable is mandatory, so that in any other application a *DoorHandle* component must contain at least this variable.

Next, we defined a type for *KessyIO* object and called it *KessyIOType*. It has a unique Id and port number. The first thing that this component does is to send a *WakeUpRequest* to a BCM. The variable and its contents are already defined in the previous component, so it can be used and does not need to be created again. *KessyIO* sends also a LFData to UID(key), a variable must be created and named and it has the data type UInt32. Since this is low frequency data, 32 bit long data type should be sufficient. In Addition *KessyIO* sends a CAN entry validation message to BCM. We created a variable with Double data type, as we are dealing with a large message size. Furthermore, a *WakeUpMethod* had to be created that is also used in other components. The method has Boolean variable as input. If the value is true the component is woken up and starts publishing, otherwise it just subscribes.

We have defined the UID key as *UIDkeyType* with unique IP address and port. The key has received a message from *KessyIO* LFData and responds by sending HFdata to BCM. A variable HFData of type UInt64 and calculated signature variable of type Byte were created in this component.

Then we defined a *BCMType* with IP adress and port number. It is a publisher and subscriber component, which first receives a *WakeUpRequest* and then, with the *WakeUp* method, the component is woken up and can publish. First of all, it sends a *WakeUpRequest* to *DoorModules*. *WakeUpRequestType* has already been defined. Then again after getting a HFData from UID key, sends to UID key a calculated challenge and signature, creating a VariableType of type Double. After receiving calculated signature from UID Key, BCM validates the data and then sends an unlock request to *DoorModules*. We then created an *UnlockRequestType* with Byte as a data type.

At the end we define a type for the last component and its variables, methods and attributes. *DoorModulesType* is a type created to instantiate the *DoorModules* component. The component is a pure subscriber class that only listens to BCM. After the component has been woken up by the BCM by sending a WakeUpRequest, it later receives an unlock request message with a defined *UnlockRequestType* and Boolean as data type. For each variable that was created, it must also contain a field with the specified data type and then the object can be fully instantiated with its attributes variables and methods.

The metamodel is now complete created (written in XML language) and can be compiled into the information model. We have compiled it with the docker container. It outputs the information model and other several files for further applications. The information model consisting a set of nodes (*Nodeset*). This *Nodeset* is a graph-based data structure that contains the content of information model, in which typed nodes are linked through typed references. Each node in the information model is assigned attributes depending on its node class (objects, variables and methods). Following attributes are obligatory for all node classes.

NodeId unique identifier of a node inside a OPC UA.

BrowseName identifies a node when browsing through the address space.

DisplayName name of the node to be displayed in a user interface.

Every node is one of the eight node types some of them are listed and may represent a variable, a method, a custom data type. The node type defines the attributes the node contains. Although information models are defined only in servers, clients are able to interpret information models, because they use the concepts of the address space model. For the application of this work PubSub meschanism has been used and an information model has been created for it, although information model PubSub is not yet completely implemented and there are very few tools that support it. It is not currently possible to use all the features of this information model, but over time tools and programs will be developed so that it can be fully used later.

4.2.4 Keyless Entry Information Model

This section explains the keyless entry use case information model in details. It is easier for the developer to understand complex applications like keyless entry use case by looking at a clear structure of the application, where the most important components, variables and methods are listed. Without information modelling of the use case it will be difficult and time consuming for the developer to understand the long C code implementation of the keyless entry use case. This is also important to ensure a good definition and understanding of this complex system. That is one reason of many why in this work information model was created.

In keyless entry use case there are five components that exchange many messages with each other in different times. Each component has attributes and methods, it involves many lines of code. It is important to create a structure for this application. The information model then gives a unified structure over keyless entry, what are there for components, what does each component have for attributes and methods and their data types on and who communicates with whom.

Figur 4.12 gives an overview of keyless entry information model. We explain in the next paragraphs the information model for each component and show the graphical notation of information model for each component. The top component *KessyType* is like a class representing the complete use case. In each keyless entry application must contain at least five components. With *HasSubType* reference *KessyType* is associated with its children. In each *KessyType* two attributes are mandatory, so each subcomponent of *KessyType* must have these attributes and assign a value to it.

- URI: each component has a unique ip address and port.
- Id: each component has a unique id.

DoorhandleType

The data type for the variable *URI* is string and for *Id* is integer. The first subtype of *kessyType* is *DoorhandleType*. When the door handle is pulled a *WakeUpRequest* is sent to *kessyIO*. A *WakUpRequestType* must be created in the information model for *Doorhandle*, which has Boolean as the data type. Then using this type, a variable *WakeUpRequest* could now be created in information model and which is a subchild of the instantiated object *Doorhandle*. Figure 4.7 shows the information model graphical notation of the *DoorHandle* component and their attributes.

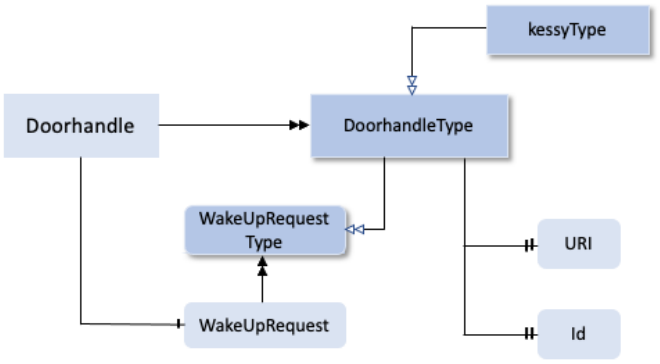

Figure 4.7: Graphic notation of *DoorHandle* component.

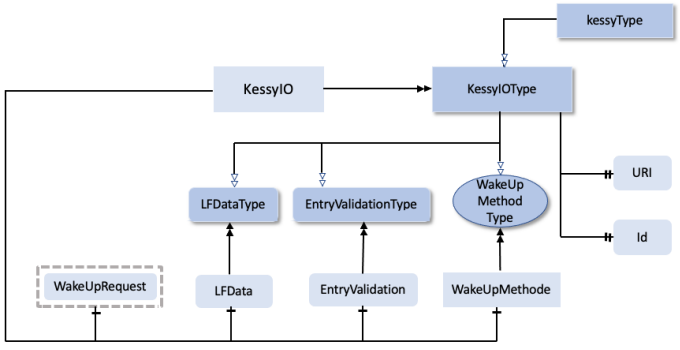

Figure 4.8: Graphic notation of *KessyIO* component.

KessyIOType

The *WakeUpRequestType* message sent to *KessyIO* should *WakeUp* the component. After *KessyIO* is woken up it sends directly a *WakeUpRequest* message to *BCM*. Figure 4.8 shows the graphical notation of the information model for the *KessyIO* component. The methods and variables with dashed rectangle indicate that a type has already been created for the variables and methods and is used in this component. Since *WakeUpRequestType* is already defined, the variable connects to BCM with a hierarchical reference *HasComponent*. Next a *low frequency data (*LFData) is sent to *UID (key)*, for this a *LFDataType* variable is created, as data type UInt32 was set. *KessyIO* sends a CAN entry validation message to *BCM* to verify the key. For this purpose a variable *EntryValidationType* of datatype UInt32 was defined. A created BCM object has at least three relationships (HasComponent) to the three created variables.

UIDkeyType

UID (key) receives a *LFData* message from *KessyIO* and then sends a *high frequency* message (HFData) to BCM. The *UID (key)* sends only data to BCM. BCM receives the *HFData* and obtains calculated signature, which the key must send. A *UID (key)* in each kessy application has subcomponents like *HFData* and SignatureKey and for the two variables *HFDataType* and *SignatureKey* were created as seen in Figure 4.9. Thereby *HFData* is set to UInt64 and for *SignatureKey* a float data type is set.

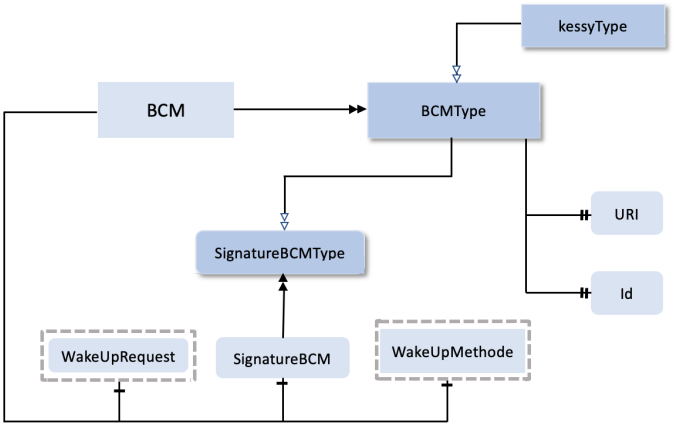

Figure 4.10: Graphic notation of *BCM* component.

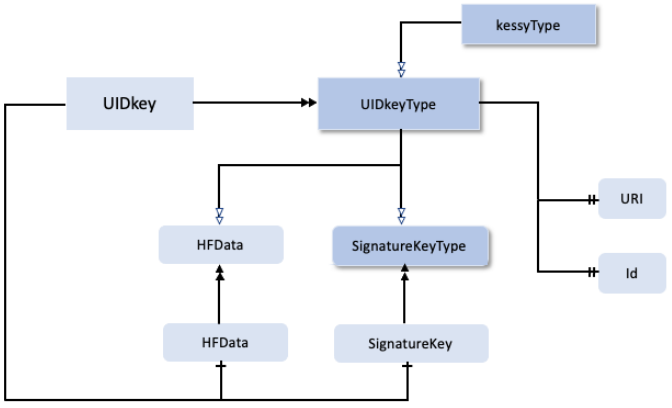

Figure 4.9: Graphic notation of *UIDkey* component.

BCMType

BCM is the largest component in keyless entry and handles most of the operations from the use case. Communicates effectively with all except *Doorhandle*. It has the most variables and methods and is therefore the largest block in the information model. It starts to work when *KessyIO* has sent a *WakeUpRequest* message to it. The sent *WakeUpRequest* has the value true and calls the method *WakeUpMethod* which wakes up the *BCM*. The *WakeUpMethod* has as in Figure 4.10 an ellipse shape with dashed recheck, since their type has already been defined and used before and does not need to be defined again in this component. The method is also related (HasSubtype) to *KessyIO* and *Doormodules*, which are woken up on request. After that the *BCM* receives *HFData* from the *UID key* and for checking if it is the right key for the vehicle it sends a created challenge and signature message to the *UID key* named in information model *SignatureBCM*. *BCM* also sends a *WakeUpReuest* message to *DoorModules*. At the end *BCM* waits for send calculated signature message von *UID (key)* and sends to *DoorModules* a *DoorUnlockRequest* via CAN. *The DoorUnlockRequestType* takes true or false as values, so it has a Boolean as data type.

DoorModulesType

The last component in keyless entry information model is the *DoorModules*. Figure 4.11 shows the information model graphical notation of the *DoorModules* component and their attributes. It is only responsible for lock and unlock the vehicle doors. Accordingly the door *UnlockRequest* via CAN the door will remain closed or will be opened. For this purpose the method *UnlockDoorModulesType* is created and from it the method *Unlock-DoorModules* with Boolean as data type is derived, which executes the request.

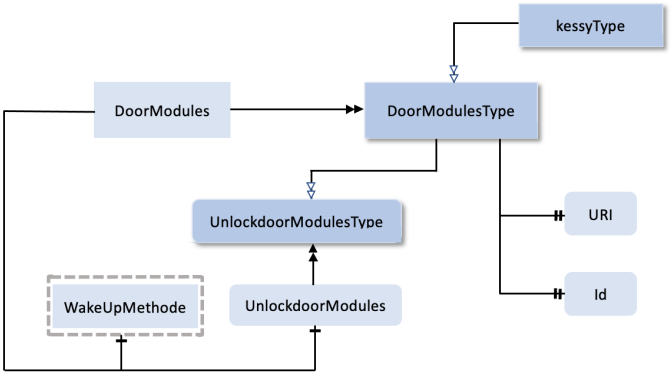

Figure 4.11: Graphic notation of *DoorModules* component.

At the end of this section we have defined the data types of the exchanged information between keyless entry components and their structure. In addition, we have mapped the exchanged information with information model as shown in Figure 4.12 and therefore we answered all research questions under **Goal 4**.

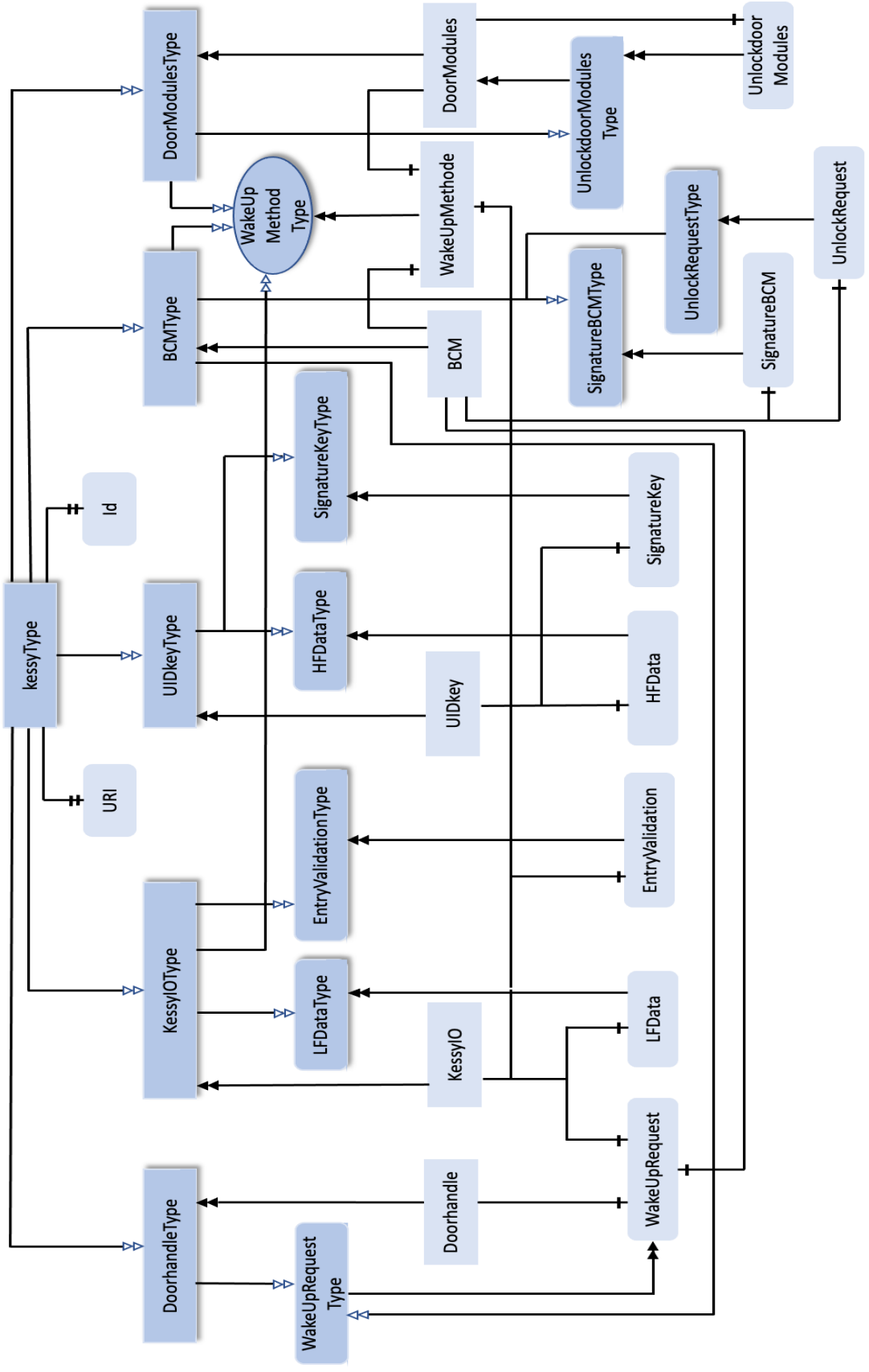

Figure 4.12: Overview of keyless entry information model.

4.3 Usability of Keyless Entry Information Model

An information model is a representation of concepts and the relationships, constraints, rules, and operations to specify data semantics for a chosen domain of discourse. Typically it specifies relations between kinds of things, but may also include relations with individual things. It can provide sharable, stable, and organized structure of information requirements or knowledge for the domain context. Accompanying the presentation of our developed structure for the keyless entry use case, we explained in the previous sections how such a model is created. The structure for our keyless entry use case is defined in an information model. It is created according to section 4.2.2.

We described in Section 4.1 all involved components of keyless entry use case with the exact description of all it's components individually and how they all interact with each other. information model gives the keyless entry use case an accurate data structure that contains the content of keyless entry system, in which typed nodes (objects, variables and mehtods) are linked through typed references. Furthermore, we created graphical notation for the use case from the information model (see Figure 4.12). It models the application structure and information exchange in keyless entry system.

It is easier for the developer to understand complex applications like keyless entry use case by looking at a clear structure of the application, where the most important components, variables and methods are listed. Without information modelling of the use case it will be difficult and time consuming for the developer to understand the implementation of keyless entry by reading the lines of code of the implementation. It gives the system a software architecture ensure a good definition and better understanding of this complex system.

What is not considered in Figure 4.5 are the last two steps *connect application* and *client validation* because these two components are currently only applicable to OPC UA client-server and not as in our case with OPC UA PubSub. But we have performed and tested these steps for a keyless entry client-server application. We have used it by exploiting another property of the information model, which is also currently usable for client-server applications. The other capability of information model is code generation but only or a number of SDK's. Our SDK open62541 supports code generation with information model.

In summary, information model is created through some steps. Creating a metamodel written in an XML language that contains the idea of the application in our case for our keyless entry use case. This metamodel contains the application structured consists of objects, variables, methods and references what exactly is exchanged between the components. In the next step we compiled the metamodel with precompiled docker container. It then creates the information model represented as a set of nodes stored in an XML schema named *NodeSet2.xml*. The compiled *Nodeset* is a graph based data structure that contains the content of information model. It outputs also several files for other applications, which will be made available later with tools. With the *NodeSet2.xml* we can then use nodeset compiler supported by open625641 and transform it to code of the application based on the data specified in the created information model.

This is then explained in detail as follows. The information model could be transformed into C source code with any supported stack or modeler. Open62541 contains a python based nodeset compiler that can transform these information model definitions into a working server. The nodeset compiler can be found in the tools/nodeset_compiler subfolder. It's not an XML transformation tool but a compiler. That means that it will create an internal representation when parsing the metamodel and attempt to understand and ver-

ify the correctness of this representation in order to generate C code [70]. The following instruction 4.1, shows how to start python compiler and generate code successfully.

Listing 4.2: Code generation from information model with python compiler.

```
$ python ./nodeset_compiler.py --types-array=UA_TYPES --
    existing ../../deps/ua-nodeset/Schema/Opc.Ua.NodeSet2.xml
    --xml keylessEntry.xml keylessEntry
```

4.4 Concept of a Publish/Subscribe Application

We have explained the components for keyless entry use case and implemented the use case in Section 4.1. In Section 4.2 we have shown how we structured the use case and defined who communicates with whom and what is exchanged between the components. In this section we discuss requirements for our prototype and in the next section how the software architecture looks like from the implementation and what we did with the implementation.

4.4.1 Requirement for our Prototype

We specified a representative use case of in-vehicle communication. We thought about, how we can implement our keyless entry use case with OPC UA PubSub. Then we implemented complex in-vehicle communication use case with OPC UA PubSub. First in Section 4.1, we defined all requirements for the use case, all involved components of keyless entry use case with the exact description of all its components individually. In keyless entry different entities communicate with each other, the communication between the components and what information is exchanged are defined according to the use case.

The vehicle *Doorhandle* first wants to publish to the vehicle antennas (*KessyIO*) that a driver has sent a request by pulling the door handle or touching it. *KessyIO* is connected to *Body control module (BCM)* via *Controller Area Network (CAN)*-Bus and notifies it. BCM checks the data (the identification of the key) and the UID key sends the important data for validation to BCM. BCM checks the sent data and sends a request to *DoorModules* to open the vehicle door.

The point now is to know, what is the data type of the exchanged information and how can it be structured. A lot of data are exchanged between the components. We were looking for a suitable data structure for keyless entry use case. An information model, similar to object-oriented programming, describes types that can be expanded and instantiated. Its semantic is one of OPC UA's strengths. But how can the information exchanged in keyless entry system be mapped to an existing information model. We have created an information model in Section 4.2 that defines what are the data types of the exchanged Information and how can it be structured.

4.4.2 Concept for a Prototype Implementation

There is a structure for OPC UA PubSub that contains all necessary elements for publisher class and for subscriber class. There are entities in OPC UA PubSub structure that are relevant for publishers for publishing the messages independent of the number of subscribers. Therefore for subscriber for receiving messages independent of the application that configures what the publisher is sending.

The entities used for our PubSub implementation are *PublishedDataset*, *WriterGroup* and *DataSetWriter*. In our model, the publisher produces messages that are sent to UDP multicast is used as transport protocol. The content of the exchanged messages is called *DataSet*.

The selection of the input for the *DataSet* messages is called *PublishedDataSet*. The selection can be a list of variable values or the fields on an *EventType*. The *DataSetWriter* is linked to a *PublishedDataSet* that defines the source of the message content. The *DataSetReader* is the counterpart to a *DataSetWriter* on the subscriber side. It also contains the *SubscribedDataSet* that defines the processing of the received data are described in detail in Chapter 5.

The methods *PublishedDataset WriterGroup* and *DataSetWriter* must each publisher include them at least once. The library open62541 are written in C. We have thought of creating a class for each keyless entry component. Each class and therefore each component is either a publisher, subscriber or publisher and subscriber together. To save many lines of code and for better clarity and understanding, we have written our keyless entry implementation in C++. We have created a publisher class that contains all publisher functions. Each publisher class in keyless entry implementation like *Doorhandle.cpp* instantiates a publisher objeckt. It must have Id, IP address and port assigned to it. Publisher create one or more *WriterGroups* with different topics. *DataSetWriter* creates a *DataSetMessage* from a *DataSet*. *DataSet* can have one or more fields with different contents and data types. This *DataSetMessage* can be a part of the *NetworkMessage*, which is created by a *WriterGroup* and then transported to the message oriented middleware. In our model the publisher sent messages to UDP multicast.

DoorHandle.cpp class is a pure publisher class. This is the only class that only publishes and does not subscribe to anything. It is only waiting for user input, in the sense that a driver with a vehicle key approaches the vehicle and door handle is pulled or touched. *DoorModules.cpp* class, which is used for lock or unlock the vehicle door is a pure subscriber class. When a publisher publishes a message and a subscriber receives the message, it should be understood by the subscriber. To be able to understand a published message a *DataSetReader* has to be created. But OPC UA PubSub implementation in library open62564 is not mature and this part is not fully implemented. We used a prototyp implementation of subscriber to be able to understand the subscribed message. The subscriber can be able then to extract *SubscribedDataSet* from a *DataSet*, which is a list of target Variables. As soon as the publisher starts to publish, the subscriber will start to receive the messages, that are in the subscription. The *DoorModules* listens on the same channel that the BCM publisher has created for certain topic and thus the subscriber receives necessary updates via the *WriterGroup*, if the value changes or the publisher publishes something new on the same topic.

The other keyless entry components, *KessyIO*, UID key and BCM are publisher and subscriber classes. They contain the functionalities of publisher and subscriber as explained above. We have created an in-vehicle communication use case in software that opens the vehicle door with OPC UA PubSub as communication protocol for the communication between keyless entry components.

A lot of data are exchanged between the components in keyless entry use case. We created an information model for our use case that defines the data types of attributes and variables, methods and references between keyless entry components. It's similar to object-oriented programming, describes types that can be expanded and instantiated and give a structure of the entire use case.

Our application in OPC UA supports the wake-up via network. Because distributed IoT devices that are powered by batteries the OPC UA protocol stack has been optimized, in which it is possible for clients to wake up, send a status message to the server and go

back to sleep without waiting for a response. This reduces the communication overhead to increase the battery lifetime and maximize energy efficiency. Additionally, with this extension, OPC UA network footprint can be scaled down to fit into different devices and transport technologies with small frame size. In our application it looks like this: first the *Doorhandle* is triggered, when the door handle of the vehicle is touched or pulled, the door handle component wakes up the *KessyIO* component which is in sleep mode like the other components. *KessyIO* is then woken up next and then wakes up *BCM* and the component in turn wakes up the *DoorModules* and when the components complete their tasks then they go into sleep mode.

It is assumed that we have real-time requirement for our application. Real-time is generally not supported in the OPC UA PubSub. The OPC UA PubSub we used is a real-time technology, but it's not time-aware. OPC UA over *Time-Sensitive Networking (TSN)* is the solution and successor to OPC UA vendor-independent technology. TSN is an upgraded network infrastructure based on benefits of Ethernet standard. With addition of latest OPC UA PubSub, TSN will be able to carry different types of industrial traffic real-time, while keeping up the performance. However, with TSN technology PubSub becomes a flexible protocol that distributes data real-time but also effectively at certain time moments when needed. Our concept provides to use TSN. However, we have not implemented it in our application, as it has to work on a low network level.

5. Implementation

Since OPC UA is nothing new, a lot of effort has already been given to the development of various applications. This chapter reviews some of the existing tools and implementations that were taken into account for the development of the keyless entry use case. Section 5.1 explains the main concepts used for the implementation of the keyless entry use case. Section 5.2 deals with the limitations of using the OPC UA library on microcontrollers.

5.1 Implementation of Keyless Entry

In this work, the applicability of OPC UA in-vehicle communication is examined. A complex use case is implemented for this issue. Communication for the use case happens with OPC UA PubSub brokerless mode. In the beginning, the communication scenario must be checked. For this purpose it was tried to get the most suitable implementation of the OPC UA stack. There are currently many open source implementations of OPC UA (see in appendix Table 7.2 and Table 7.3). All of them are maintained and developed by companies and research institutes. Amongst some of the implementations, the ones worth mentioning are open62541 written in C programming language. Open62541 implements the latest version of OPC UA with a PubSub extension. The library is well-maintained with lots of contributors and also organizations (e.g. Fraunhofer) and universities (RWTH Aachen, TU Dresden) contributing to it.

The work was started by first getting a client-server communication up and running, to test the library and take the first steps. The process of setting up the library open62541[1] goes through a couple of steps[2]. OPC UA client-server is executable on many *General-Purpose Operating Systems (GPOS)* and *Real Time Operating Systems (RTOS)*. There are a lot of features[3]can be optionally enabled using the build options. A flag was used to generate a single open62541.h header file instead of multiple single header files, named „UA_ENABLE_AMALGAMATION=ON“. For Testing client-server communication[4][5] only open62541.h is imported as header. But it is not recommended to turn on the amalgamation option.

[1] `https://github.com/open62541/open62541`
[2] `https://open62541.org/doc/current/installing.html`
[3] `https://open62541.org/doc/current/installing.html`
[4] `https://open62541.org/doc/current/tutorial_client_firststeps.html?highlight=myclient`
[5] `https://open62541.org/doc/current/tutorial_server_firststeps.html?highlight=myserver`

```
$ git submodule update –init –recursive
$ mkdir build && cd build
$ cmake DBUILD_SHARED_LIBS=OFF
    DUA_BUILD_EXAMPLES=ON
    DCMAKE_BUILD_TYPE=Debug
    DUA_ENABLE_PUBSUB=ON
    DUA_ENABLE_PUBSUB_
    ETH_UADP=ON
    DUA_ENABLE_PUBSUB_
    INFORMATIONMODEL=ON
    DUA_NAMESPACE_ZERO=FULL ..
$ make
$ sudo make install
```

Table 5.1: Important *Cmake* options for a PubSub installation.

publish/subscribe (PubSub) extension for OPC UA enables fast and efficient 1:m communication. This part is still work in progress. The PubSub extension can be used with broker based protocols like MQTT and AMQP or brokerless implementations like UDP multicasting. For the use case of this work PubSub brokerles was applied. It's worth to mention that PubSub is relatively new so even in the open62541 and in other commercial implementations, PubSub is still under development.

For OPC UA PubSub, however, no official certification is currently offered by the OPC Foundation [71]. As this version and other versions do not yet include any PubSub functionality [72]. An early prototypical PubSub extension from open62541 is used in this work. Currently PubSub can only be compiled in Linux. Options must be enabled using the build options in Table 5.1. It is not straight forward to get the open62541 OPC UA stack up and running on an embedded system even if FreeRTOS and Lightweight IP (lwIP) is supported for client server communication mechanism. RTOS is an open-source real-time operating system for microcontrollers. lwIP is a widely used open-source TCP/IP stack designed for embedded systems [73]. There are several examples to test open62541 library on microcontrollers but currently they are only runnable with client server. Some of these are:

- Example project for compiling open62541 on an Arduino. This specific project uses an Adafruit ESP32 microcontroller with a WLAN module[6]. It explains in detail how to build the library on Arduino and which software and hardware dependencies are required.
- Example for an ESP32 Microcontroller. This repository[7] shows an exemplary use of the open62541 OPC UA server implementation on a ESP32 microcontroller. This implementation is used and tested from open62541 developer.

So far now PubSub can not work on microcontroller and works only in Linux OS. It is written in the CMakeLists.txt file[8] that many files are not built when installing the library on a non Linux OS and PubSub can not work without these files. It can work on *Raspberry Pi* and other minicomputers where Linux runs on it. There are other works that has applied PubSub on minicomputers e.g. [72, 74, 75]. It has been tried to enable the amalgamation feature as in client server but this feature is not supported in PubSub.

[6] `https://github.com/pro/open62541-arduino`
[7] `https://github.com/Pro/open62541-esp32`
[8] `https://github.com/open62541/open62541/blob/master/CMakeLists.txt`

Many RTOS are not full Operating System (OS) in the sense that Linux is, in that they comprise of a static link library providing only task scheduling. The open62541 library should be linked to the application using a framework or tool to produce a single executable file that a microcontroller OS can perform.

Bevor to create PubSub connection, there are components should be created for possibility of connection with PubSub. Figure 5.1 shows an overview of the components have to be built to create a PubSub connection. After the PubSub connection was established, some steps have to be done, before the publisher can actually start to publish. The *WriterGroup*, *DataSetWriter* and *PublishedDataSet* components define the data acquisition for the *DataSets*, the message generation and the sending on the publisher side.

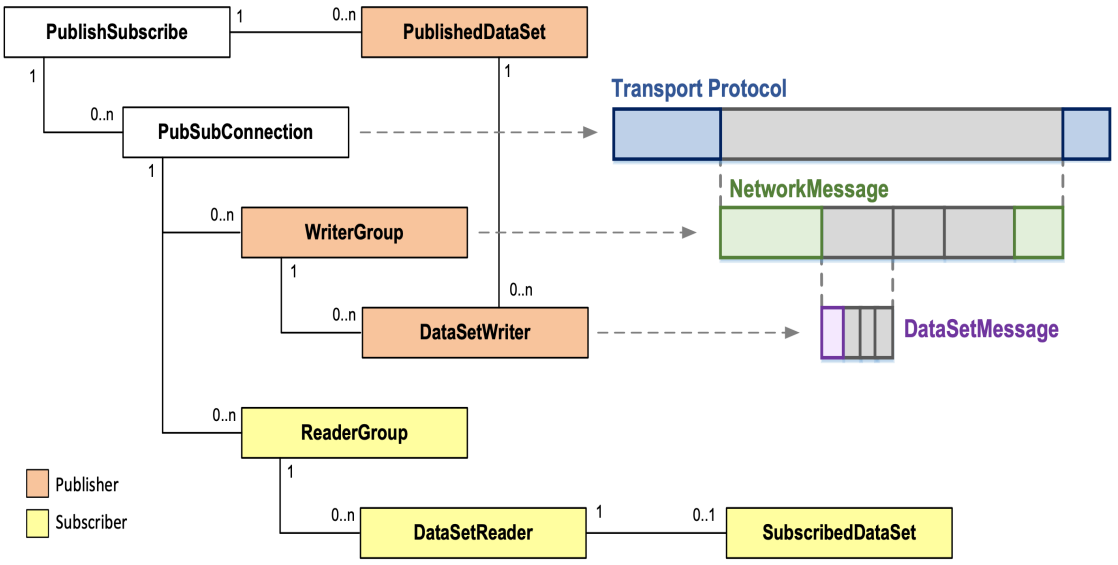

Figure 5.1: PubSub component overview [25].

At first, the PublishedDataSet (PDS) has to be created. It contains the *DataSetMetaData* describing the content of the *DataSets* produced by the *PublishedDataSet*, the corresponding data acquisition parameters and the configuration of the data collector, which both publisher and subscriber to understand the messages. All other PubSub elements are directly or indirectly linked with the PDS. The following instructions Listing 5.1 show how to create a PDS for keyless entry use case open62541.

Listing 5.1: PublishedDataSet handling.

```
1 static void addPublishedDataSet(UA_Server *server) {
2     UA_PublishedDataSetConfig publishedDataSetConfig;
3     memset(&publishedDataSetConfig, 0, sizeof(
          UA_PublishedDataSetConfig));
4     publishedDataSetConfig.publishedDataSetType =
          UA_PUBSUB_DATASET_PUBLISHEDITEMS;
5     publishedDataSetConfig.name = UA_STRING("Demo PDS");
6     UA_Server_addPublishedDataSet(server, &
          publishedDataSetConfig, &publishedDataSetIdent);\emph
          {}
7 }
```

Secondly, all publishers are added to a *WriterGroup*, which its parameters such as the priority of the message, publishing interval and security settings are necessary for creating

a *NetworkMessage*. Some of these parameters are used for creating the *DataSetMessages*. Each *WriterGroup* can have one or more *DataSetWriters*. *DataSetWriter* creates a *DataSetMessage* from a *DataSet*. *DataSetWriter* parameters are necessary for creating *DataSetMessages*. Each *DataSetWriter* is bound to a single PDS. A PDS can have multiple *DataSetWriters*. This *DataSetMessage* can be a part of the *NetworkMessage*, which is created by a *WriterGroup* and then transported to the message oriented middleware. The description of published variables is named *DataSetField*. In a *DataSet* there are one or more fields. Each field contains the selection of one information model node. The field has additional parameters for the publishing, sampling and error handling process. In each field a value can be stored from data types like Double, Float, UInt32, etc. Figure 5.2 shows how a transport message in OPC UA is created from *DataSetMessageField* to *TransportProtocol*. A field also contains useful information about the value stored in it. *DataSet* contains a header and one or more fields and from this a *DataSetMessage* is created. In a *PubSubConnection* there are one or more *WriterGroups*. A publisher can have different *WriterGroups* that are equal to understand with tokens in the PubSub with broker. Each group can create a content and when a subscriber subscribes to a particular publisher then only the subscribed content is published to him. Then the *DataSet* belonging to the *WriterGroups* is packaged in a *NetworkMessage* with the headers with the necessary parameters and from that a *Transport Protocol* OPC UA message is ready to be sent. Now the publisher starts publishing.

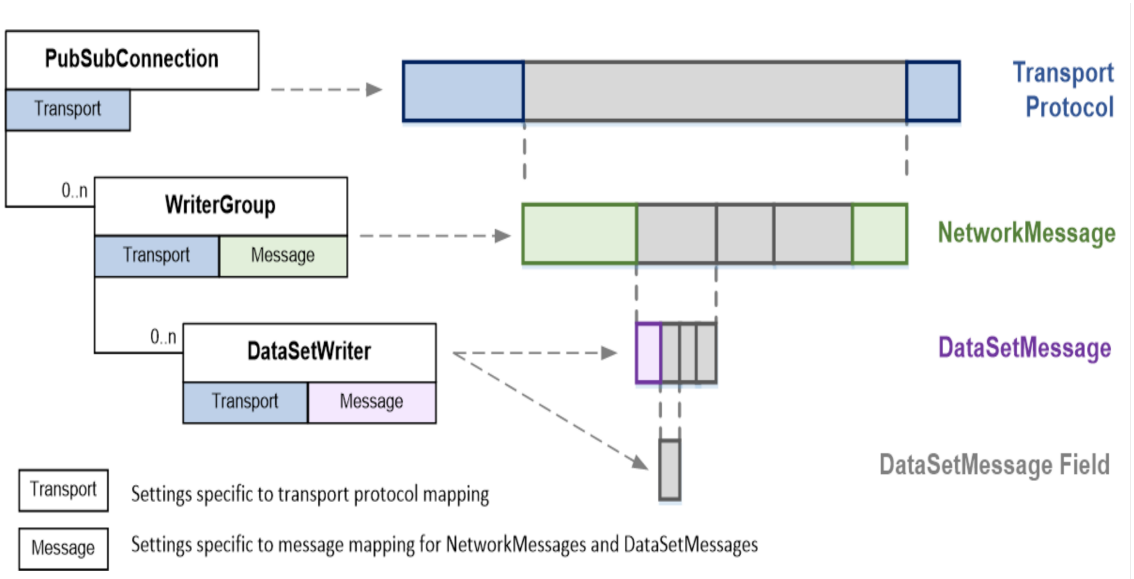

Figure 5.2: PubSub message overview [25].

The following listings show the implementation of *WriterGroup* in Listing 7.1, DataSet Listing 5.2 and DataSetField Listing 5.3. These methods must be created in each publisher. For creating several *WriterGroup*, *DataSe*t and *DataSetField* the methods are called several times.

Listing 5.2: Add DataSet to WriterGroup.

```
1 addDataSetWriter(UA_NodeId publishedDataSetIdent, UA_NodeId
    &writerGroupIdent, UA_Int16 dataSetWriterId) {
2       UA_NodeId dataSetWriterIdent;
3       UA_DataSetWriterConfig dataSetWriterConfig;
4       memset(&dataSetWriterConfig, 0, sizeof(
          UA_DataSetWriterConfig));
```

```
5           char name[] = {"Demo DataSetWriter"};
6           dataSetWriterConfig.name = UA_STRING(name);
7           dataSetWriterConfig.dataSetWriterId =
                dataSetWriterId;
8           dataSetWriterConfig.keyFrameCount = 10;
9           UA_Server_addDataSetWriter(server, writerGroupIdent,
                publishedDataSetIdent,
10                                      &dataSetWriterConfig, &
                                            dataSetWriterIdent);
11
12  }
```

Listing 5.3: Add DataSetField to DataSet.

```
1   addDataSetField(UA_NodeId &publishedDataSetIdent, const
        UA_NodeId &varNodeId) {
2           UA_NodeId dataSetFieldIdent;
3           UA_DataSetFieldConfig dataSetFieldConfig;
4           memset(&dataSetFieldConfig, 0, sizeof(
                UA_DataSetFieldConfig));
5           dataSetFieldConfig.dataSetFieldType =
                UA_PUBSUB_DATASETFIELD_VARIABLE;
6           char name[] = {"Server localtime"};
7           dataSetFieldConfig.field.variable.fieldNameAlias =
                UA_STRING(name);
8           dataSetFieldConfig.field.variable.promotedField =
                UA_FALSE;
9           dataSetFieldConfig.field.variable.publishParameters.
                attributeId = UA_ATTRIBUTEID_VALUE;
10          dataSetFieldConfig.field.variable.publishParameters.
                publishedVariable = varNodeId;
11          UA_Server_addDataSetField(server,
                publishedDataSetIdent,\emph{}
12                                      &dataSetFieldConfig, &
                                            dataSetFieldIdent);
13  }
```

Subscriber must contain a *ReaderGroup*, *DataSetReader* and *SubscribedDataSet*. A Subscriber can be a client, a server or some other application, that understands *DataSet* configuration. *ReaderGroup* is used to group a list of *DataSetReaders* and contains a few shared settings for them. The *NetworkMessage* related filter settings to ensure that the subscriber only receives the content that he has subscribed, are on the *DataSetReaders*. *DataSetReader* parameters represent settings for filtering of received *NetworkMessages* and *DataSetMessages* as well as settings for decoding of the *DataSetMessages* of interest. The *DataSetReader* extract *SubscribedDataSet* from a *DataSet*, which is a list of target variables, with the aid of all this information. The subscribed message is then received after that. *DataSetReaders* creates a *ReaderGroup* similar to a *WriterGroup*. As soon as the publisher starts to publish, the subscriber will start to receive the messages, that are in the subscription. In keyless entry use case, some components are publishers or subscribers or publishers and subscribers together in one C code class.

These were important components in PubSub brokerless that enable communication and data exchange between components. Hybrid class were the components of publisher and

subscriber are implemented together in one C code class. Another important component for PubSub communication with UDP multicasting is *UA_Server* represented as the root of the components in Figure 5.3. *UA_Server* has a unique IP ddress and port and contains one or more *PubSubChannels*. Multiple servers could be created for the keyless entry use case or multiple channels with unique IP addresses. In this work a server was created for each publisher with IP address and port. Subscribers simply listen on this port and consume the *NetworkMessages* according to their subscriptions. In this section we answered research questions **RQ 5.1** and **RQ 5.2**. They are about, which implementation we took and how we implemented our keyless entry use case in software. The next section answers the research question **RQ 5.3**, whether it is possible to run the implementation on real hardware.

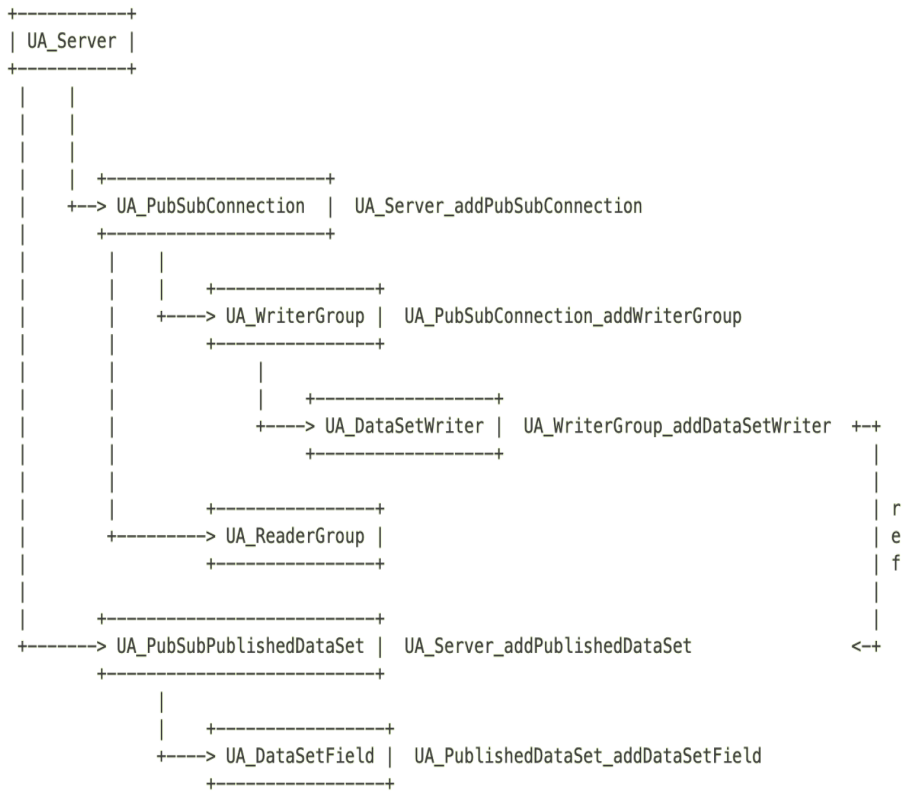

Figure 5.3: Overview of all PubSub components.

5.2 Limitations of OPC UA Library on Hardware

This section outlines the limitations of the used library and explains why, therefore, our implementation can currently not be used on microcontrollers with constraint hardware resources. In general for our implementation of the keyless entry use case there is no specific prerequisite for the device to run it. The only thing we need is network connection with other keyless entry components and some memory and CPU utilization described in more detail in Chapter 6. It is important to show what hardware the keyless entry implementation can run on and whether the implementation can run on low performance hardware. Most previous work with OPC UA PubSub have built their testbed only on *Raspberry Pi* with Linux [72, 19, 75].

For purpose of our implementation of keyless entry, it was tried to get the most suitable implementation of the OPC UA stack. We selected the latest version of the open source of OPC UA with a PubSub extension. It is called open62541 stack. It implements the latest version of OPC UA with a PubSub extension. OPC UA PubSub extension has

shown that with open62541 a usable program library exists, which we have therefore chosen for our investigation. However, the brokerless mode of OPC UA PubSub is still under development. The publisher structure itself is quite complete only the subscriber structure is not yet fully implemented according to the specified structure, which is why adjustments to the program code on the subscriber side were necessary.

For running our keyless entry use case some software prerequisites are required. This is also a reason why it is not possible to run OPC UA PubSub with the open62541 stack and other OPC UA stacks on microcontroller. For running these use case, we required *cmake*, python (<= 2.7) and a compiler (*MS Visual Studio*, *gcc*, *clang* and *mingw32* are known to be working). Our testbed runs on a virtual machine with Ubuntu 18.04 installed. We would have to run Linux on our virtual machine, because in the library in *Cmakefile.txt* stands that open62541 PubSub can only run on Linux systems. It cannot currently run on any other operating system or real-time operating system. We tried to get our implementation to run on other operating systems. However, in open62541 the most important files necessary for PubSub are not created and therefore it is currently only executable on Linux. With the current prototype we would also have to make some adjustments for the implementation to be done to run our application on it. Currently in OPC UA PubSub *AMALGAMATION* flag must be set to off and therefore PubSub currently doesn't work with amalgamated files. This flag was used to generate a single open62541.h header file instead of multiple single header files. On microcontrollers where GPOS is not running on it, such files are required. So on RTOS one application must run at a time. OPC UA library should be linked with our application code to produce a single executable that the system boots directly. But currently we can't create an executable file with the library because PubSub doesn't work with amalgamated files.

6. Evaluation

In this chapter we evaluate the keyless entry use case in terms of resource requirements. Section 6.1 describes two testbeds to evaluate the OPC UA performance and to implement the keyless entry use case. Section 6.2 analyses the UDP packet overhead exchanged between the keyless entry components. In Section 6.3 we present the results of local resource consumption (RAM, CPU) of the the keyless entry components. In Section 6.4 we investigate the results of the communication latency measurements by exchanging data with OPC UA PubSub and client/server. In Section 6.5 we discuss the aspects of this thesis and vacancies like hardware limitation, wake-up via network and real-time capability.

6.1 Description of Testbeds

This section contains a description for two testbeds. The first testbed is about the comparison of the OPC UA communication modes client/server and PubSub to evaluate the OPC UA performance and investigates the communication latency. The second testbed is the implementation of the keyless entry use case to evaluate the keyless entry use case in terms of resource requirements. For that, we use an open62541 library, which implements OPC UA PubSub and client/server.

6.1.1 First Testbed

It describes the first testbed that is used for latency and performance measurements. We conducted measurements to evaluate the OPC UA performance using the two different communication channels client/server subscription and publish/subscribe using UDP multicast UADP. For this testbed we made an experiment to measure the latency when data are exchanged between OPC UA client and server without packet loss and another time with packet loss as well as between OPC UA publisher and subscriber extensions in order to evaluate the performance for the different communication channels.

The communication between client/server and publisher/subscriber are located on the same device in the network. We show in Figure 6.1 and in Figure 6.2 two configurations for our evaluation setups. The difference between both figures, we have in the first figure a TCP connection between client/server and in the second figure a UDP connection between publisher/subscriber. In our testbed we have a *MackBook Pro* laptop that is connected to

a virtual machine in which our client/server and publisher/subscriber configurations are set up. The test application in Figure 6.1 with a TCP connection between exactly one server and one client and in Figure 6.2 UDP connection with exactly one publisher and one subscriber are implemented in OPC UA. We used for the test application open source OPC UA implementation open62541 in C programming language. This was due to the big open source community and the C99 programming language, which is compatible with most operating systems and compilers for embedded devices.

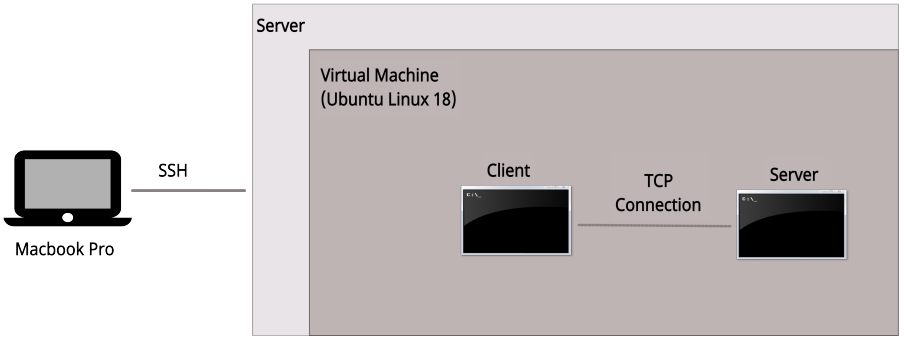

Figure 6.1: Evaluation client/server setup.

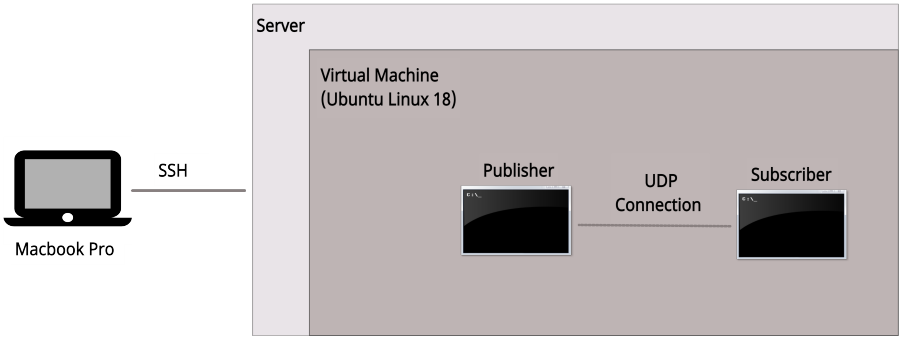

Figure 6.2: Evaluation publisher/subscriber setup.

The test application is compiled on Linux, because open62541 supports PubSub and needed networking extensions only on Linux. Our testbed runs on a virtual machine with Ubuntu 18.04 installed. The CPU consists of a single core with a 2.5 GHz processor clock rate and 16 GB RAM are available.

We have written a timestamp in packet on the server or publisher side and have evaluated the timestamp on the client or subscriber side. We run for the latency three measurements each for client/server and publisher/subscriber. Each measurement took five minutes and we sent one message per second. The data was then collected and written into csv files.

6.1.2 Second Testbed

It contains a description for the second testbed that is used to implement the keyless entry use case with OPC UA. In this testbed we have performed measurements of UDP packet overhead and performance measurements for performance (CPU utilization and memory usage) of the keyless entry components.

Figure 6.3 shows the second testbed in this work. It is similar to the first testbed in the previous section. We have a laptop as can be seen on the left side of the figure. It

is connected to a virtual machine where all our implementations in this work have been implemented. This testbed runs on the same virtual machine like in the previous testbed with Ubuntu 18.04 installed. The CPU consists of a single core with a 2.5 GHz processor clock rate and 16 GB RAM are available. It is important to mention that open62541 currently supports PubSub and needed networking extensions only on Linux.

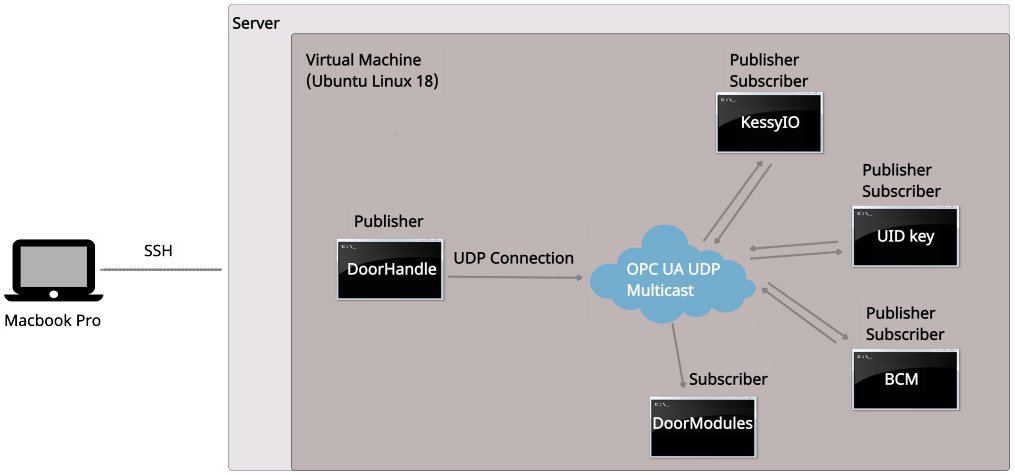

Figure 6.3: Testbed for the implementation of keyless entry use case.

We have used OPC UA PubSub brokerless for the communication between the keyless entry components. OPC UA PubSub communication pattern using UDP with OPC UA binary data encoding represents a brokerless middleware. The publisher sends its data to a predefined multicast address, while the subscribers listen to this network address. The measurement starts when the publisher sends its data and stops after the subscriber receives the published message. The brokerless version of PubSub, which is also known as multicast using *Writer*- and *ReaderGroups* messages can be published and subscribed as described in Chapter 5. The communication between the publisher and subscriber using MQTT requires more time to exchange a message than the UADP variant. One reason is the delay introduced by the broker to process and forward the message. The UDP-based communication channel does not require this processing time as there is no broker in between.

In this testbed we have five software components *DoorHandle*, *KessyIO*, *BCM*, *UID key* and *DoorModules* joining the UDP multicast group and thus have their own IP addresses and ports. Each of these five software components represents the keyless entry components described in detail in Chapter 4. Each of these components was implemented with C++ and they communicate with each other with OPC UA using open62541 stack. As shown on the right side of the Figure 6.3, each component is implemented as a publisher or subscriber or both publisher and subscriber together. We then prepared the results with the help of scripts written in Linux and wrote them in csv files. After that we wrote python scripts to evaluate and plot the results, which will be shown in the next sections.

6.2 Measurements of Packet Overhead

OPC UA PubSub protocol is considered from a theoretical point of view by looking at the overhead for transmitted package. Every protocol needs some kind of package header for each data it transmits as a payload of the UDP package so the remote side knows which kind of data is sent. This package header adds additional overhead to each data message

transmitted on the network and thus limits the possible maximal bandwidth which can be reached.

The *UA Datagram Protocol (UADP)* header formats in Figure 6.4 for both *NetworkMessages* and *DataSetMessages* were designed to be flexible and to support different use cases by enabling or disabling individual fields within headers. A UADP *NetworkMessage* header contains many fields[1], the most important of which are *PublisherId* and *WriterGroupId* to identify the *WriterGroup*. Each *NetworkMessage* contains the same number of *DataSetMessages*. The sequence of the *DataSetMessages* within a *NetworkMessage* is the same in every publishing interval. UADP messages may be signed to ensure integrity. In this case the *SecurityHeader* and the *Signature* have to be added to the message. As the library for PubSub communication is not mature, signing and encryption features were disabled in this work.

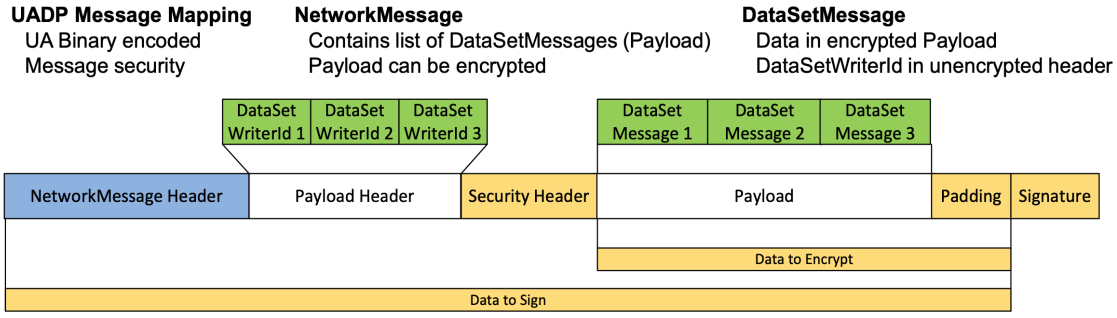

Figure 6.4: OPC UAPubSub header [76].

In addition there is the so called OPC UA *NetworkMessage* which forms the payload of the UDP datagram, each *NetworkMessage* having the OPC UA specific header and footer and containing one or more *DataSetMessages*, which in turn have so called *DataSetMessage* fields. In our case, the *NetworkMessage* contains only one *DataSetMessage*. Each *NetworkMessage* provides unencrypted data in the *NetworkMessage* header to support identifying and filtering of relevant publishers, *DataSetMessages*, *DataSetClasses* or other relevant message content. Figure 6.5 shows the used UADP *NetworkMessage*. The NetworkMessage has a protocol header overhead of 21 bytes. In addition, a *DataSet* must be created that contains the fields and has a header of 8 bytes. Because the encoding of the Variant type is used, an additional one byte is needed to classify each *DataSetField* type in the *NetworkMessage*. For instance, to send message of user data with empty *DataSet* the *NetworkMessage* is 30 bytes long and for each additional field in *DataSet*, data type size plus 1 byte is added.

OPC UA defines 25 built-in data types in Table 7.3 that form the base of the DataType-node hierarchy. Every data type that is based on the built-in data types has a given size that changes the *NetworkMessage* size, when create a field and define the field data type. The following tables show the protocol payload size that is passed from all publishers who have published in the keyless entry use case. For OPC UA, it is evaluated the overhead for a *DataMessage* sent from the publisher to the subscriber without encryption, which would add additional overhead. Note, that these values include UDP header sizes and are therefore influenced by the Ethernet frame size, as the middleware payload size is independent from this if the frames are split at a lower layer.

[1] https://reference.opcfoundation.org/src/v104/Core/docs/Amendment6/readme.htm

The following values are entered in each of the following tables. First of all, the package size in bytes transmitted as UDP, number of fields in a *DataMessage*, *Field* type and *DataMessage* in bytes. The difference between the package size and the protocol payload shows the protocol overhead. The second column summarizes the total bytes for the connection.

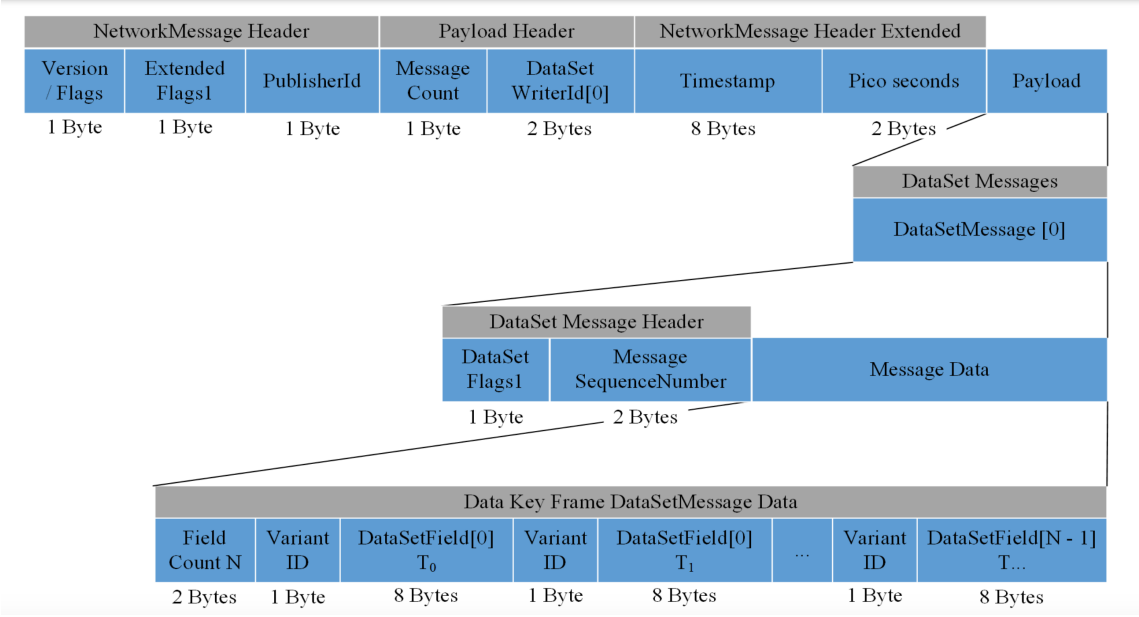

Figure 6.5: An overview of the used UADP *NetworkMessage* [77].

Additionally, the values are verified using *Wireshark* in Figure 6.6. The lower arrow with the *NetworkMessage* header points to the size of an empty *NetworkMessage* header without value (only the header information). The upper arrow points to a *NetworkMessage* header 30 bytes plus one byte for each field and 8 byte for the content, as it is *VariableType* Double. Further, headers of UDP protocol for communication is 72 bytes in total for first message without fields and 81 bytes for second message. Furthermore, with *Wireshark*, further information could be taken, such as IP address and port of source and destination and further details about the sent messages as well.

DoorHandle				
No.	Frame Size	No. of Fields	Field type	User Data
1	74	1	Boolean	32

Table 6.1: Overview of DoorHandle published messages.

KessyIO				
No.	Frame Size	No. of Fields	Field type	User Data
1	81	1	Double	39
2	81	1	UInt64	39
3	81	1	Float	39

Table 6.2: Overview of KessyIO published messages.

UID key				
No.	Frame Size	No. of Fields	Field type	User Data
1	81	1	Double	39
2	81	1	UInt64	39

Table 6.3: Overview of UID key published messages.

BCM				
No.	Frame Size	No. of Fields	Field type	User Data
1	74	1	Byte	32
2	77	1	UInt32	35
3	74	1	Boolean	32

Table 6.4: Overview of BCM published messages.

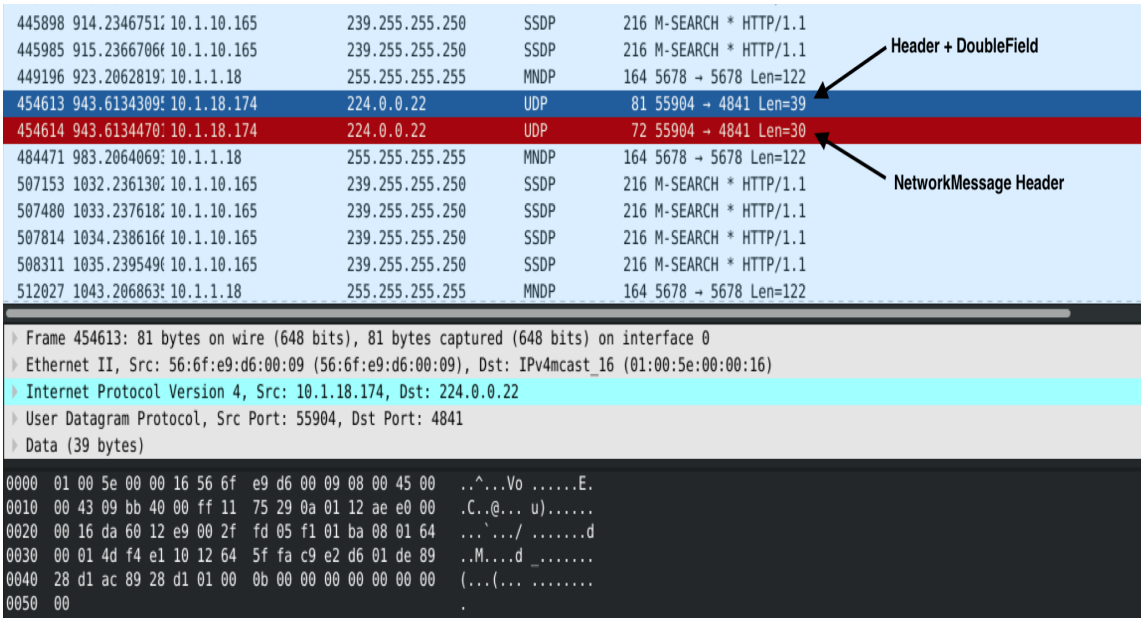

Figure 6.6: Capturing UDP packets with *Wireshark*.

6.3 Performance Measurements

This section presents our performance measurements results for CPU utilization and memory usage of the keyless entry components and OPC UA publish/subscribe and client/server. First, the results of the first testbed are evaluated then the second one. All keyless entry components, publish/subscribe and client/server were implemented using OPC UA open62541 SDK. As this version does not yet support complete PubSub functionality, we obtained and used an early prototypical PubSub extension from open62541.

6.3.1 Investigation of CPU & RAM Consumption of Keyless Entry Components

We want to know exactly how much memory usage and CPU utilization each component of the keyless entry consumes. This is important in order to run the keyless entry implementation on hardware later. It is important to show what hardware the keyless entry implementation can run on and whether the implementation can run on low performance hardware. Table 6.5 shows the CPU utilization in percent and memory usage in kilobyte (KB) for each component of the keyless entry. The first column shows a list

of keyless entry components. The second column shows the CPU utilization in percent for each component. In the third column, the memory usage entries are entered in kilobyte.

We developed keyless entry components in C that use OPC UA PubSub functionality to simulate keyless entry use case. We used tools pre-implemented in Linux to monitor the performance metrics (CPU utilization and memory usage) to be evaluated during the experiment. CPU utilization is the usage of time a CPU is occupied running processes. For a CPU, it's the busy time is spent processing program instructions and not the time that CPU are running the idle thread. To get the results of the CPU utilization, there are different processes monitoring tools. For our results CPU utilization is computed by the operating system Linux *top* or *htop*. It reports the average CPU utilization over 1, 5, and 15 minutes. We see in Table 6.5 that the *DoorHandle* is implemented as a pure publisher and publishes only one topic that utilizes CPU of 3%. *DoorModules*, which was only implemented as a subscriber, has a similar CPU utilization of 3% as *DoorHandle*. These two components have a lower CPU utilization than the rest of the components.

Component	CPU Utilization	Memory Usage (KB)
DoorHandle	3.0%	13.564
KessyI/O	3.3%	13.640
UID (key)	3.3%	13.648
BCM	3.3%	13.644
DoorModules	3.0%	1.260

Table 6.5: Shows CPU and memory utilization of the keyless entry components.

KessyIO, *UID (key)* and BCM have a higher CPU utilization than *DoorHandle* and *DoorModules* of 3.3 %. Since it is a matter of publishers and subscribers subscribing to certain topics and publishing several topics to subscribers, the CPU utilization is higher. This means that more *WriterGroups* and therefore topics are created, which utilize the CPU more than the other two components. Now, we want to look at memory usage for each individual components. Memory usage is the physical amount of memory a process is consuming in *kilobyte* (KB). *DoorModules* has the lowest memory usage of all other components because it does not contain *WriterGroups* and only subscribes. It has about 1.260 KB of memory usage. *DoorHandle* has a lower memory usage than *KessyIO*, UID (key) and BCM, which have similar values to each other, as *DoorHandle* creates only one *WriterGroup* and thus publishes one topic. The last components *KessyIO*, UID (key) and BCM have similar values. Several *WriterGroups* are created in these components. What also plays a role in memory usage is the message size which also depends on the data types of the messages.

6.3.2 Investigation of CPU & RAM Consumption of Publisher/Subscriber and Client/Server

In the following section, we describe a performance model which we created in Section 6.1.2. Such a performance model can be integrated in development processes in order to evaluate whether client/server subscriptions or publisher/subscriber are feasible for a given use case or a specific device. To improve our understanding of OPC UA performance further, we developed test programs that use OPC UA functionality to simulate client/server and publisher/subscriber respectively. In this section we show how much memory usage and CPU utilization our test programs consume. With that developers can size hardware environments to minimize costs while providing an appropriate performance.

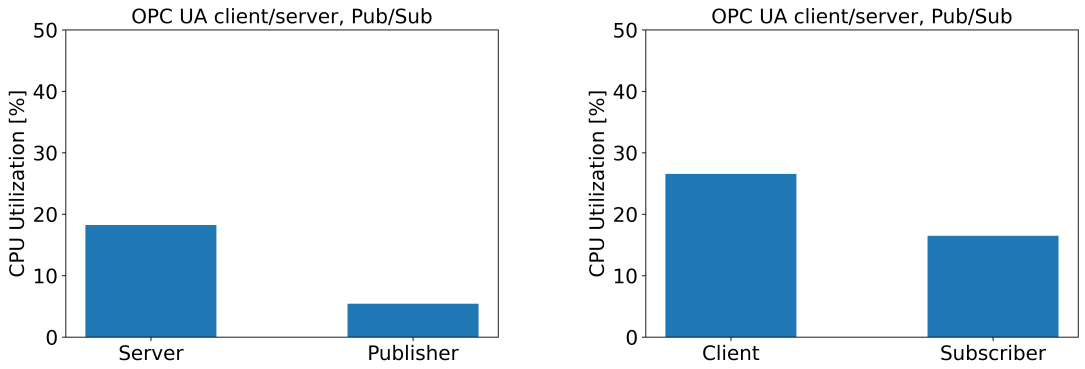

Figure 6.7: Comparision of CPU utilization for OPC UA modes: PubSub and client/server.

We tried to send as many packets as possible to keep the CPU utilized and see if client/server still consumes more or less CPU than publisher/subscriber. Within one millisecond, three packets were sent in publisher/subscriber mode and in client/server mode. We looked at various tools that show the average of CPU usage. Users with more powerful hardware could achieve even higher server or publisher throughputs, while users with much less powerful hardware could run into bottlenecks. We did not conduct a comparison of further OPC UA SDKs for PubSub performance, so our results may not be easily generalized to applications using different SDKs. We used the Linux tool *top* and *htop* to measure the average CPU utilization. The CPU utilization is calculated based on the time the CPU spends executing any of the program's processes and the total elapsed time.

Figure 6.10 on the left shows the average of CPU utilization of the OPC UA for a server and for a publisher. The server CPU utilization using client/server communication is higher than for this case using PubSub communication. The average of CPU utilization for a server in client/server mode is about 18.24%. The publishing OPC UA server and publisher can send 3 packets per millisecond via TCP by client/server and via multicast UDP by pub/sub on our testbed, before the CPU is saturated. The average of CPU utilization for a publisher in pub/sub mode is about 5.43%. In the client/server communication, the server CPU utilization was higher than for PubSub communication with the same packet rate.

Figure 6.10 on the right shows the average of CPU utilization of the OPC UA for a client and for a subscriber. A client has the highest CPU utilization compared to server, publisher and subscriber. It has an average of CPU utilization about 26.55%. A subscriber in PubSub communication mode has an average of CPU utilization about 16.48%. In the client/server communication, the server and client CPU utilization were higher than for publisher and subscriber in PubSub communication mode with the same packet rate.

6.4 Investigation of Network Latency

This section investigates the results of the communication latency measurements by exchanging data with OPC UA PubSub brokerless and compared to the client/server based communication with and without packet loss. In this experiment, we measure the latency between one client and one server as well as between one publisher and one subscriber in order to evaluate the performance for the different communication channels. The communication partners are located on the same local device (a virtual machine) as described in Section 6.1.2.

For the subscription via the client/server pattern, the measurement starts when the subscription request of the client is confirmed by the server and therefore a session is created. The OPC UA server responds immediately after subscription with a first notification containing the current value of the subscribed variable. The content of the network message is the timestamp sent on the server side and captured and evaluated on the client side. The measurements stops when the client receives a notification message from the server.

In case of OPC UA publish/subscribe pattern with UADP, we measure the latency between subscriber and publisher using UDP as with OPC UA binary data encoding. Since this communication represents a brokerless middleware, the publisher sends its data to a predefined multicast address, while the subscribers listen to this network address. The measurement starts when the publisher sends its data (provides its data to the network), while the subscriber is the consumer of this information. Both entities are defined to be decoupled. The timestamps created in this scenario are basically the same to the former test with client/server pattern. The measurements stops after the subscriber receives the published message.

We let each measurement run for half an hour. After half an hour the measurement stops automatically and then the collected data is plotted with python. Figure 6.8 shows latency to publisher/subscriber shown on the left and client/server shown on the right. We can observe that the median latency for publisher/subscriber is 36 and the mean is about 41.31 microseconds. For a client/server the median latency is 79.0 and the mean is about 80.14 microseconds. The latency to exchange a message using the client/server pattern is quite high compared to the time required to send the same message using one of the publish/subscribe channels. There are several outliers in the PubSub as well as in the client/server. These can be explained by something strange happening in the background or the timing of the network has varied. Furthermore, the outliers with client/server have higher latencies than with PubSub. We can conclude that the time required to receive the published message is higher in client/server mode than in publisher/subscriber mode.

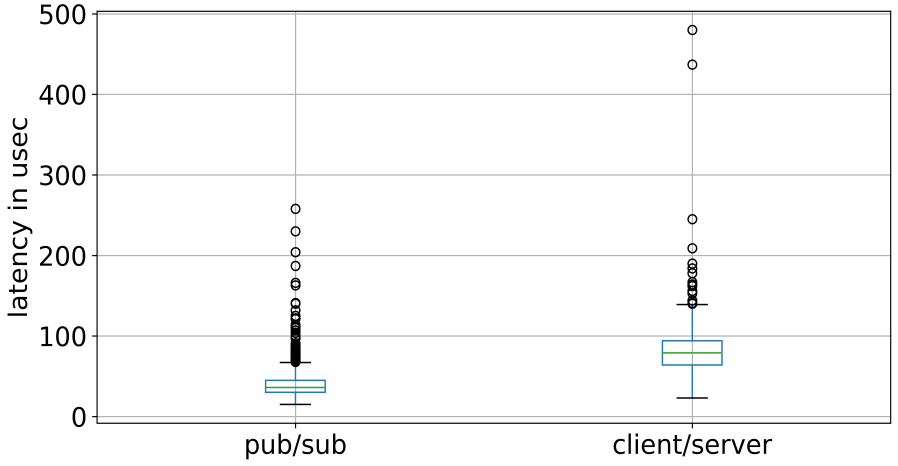

Figure 6.8: OPC UA PubSub and client/server latency measurements.

Figure 6.9 show the box plot for the PubSub, client/server and client/server with packet loss of the message being sent from the server or publisher to the client or subscriber side. This plot is similar to the previous plot but with additional 10% packet loss to the client/server communication mode. The plot is limited on the y-axis to 1 milliseconds. This

means that there are many outliers on the client/server packet loss side that are not shown in this plot so that the other more important values can be better viewed. It is interesting to see how much latency the client/server gets when a packet is dropped and resent. The measurement ran for half an hour as in the previous plot and one packet was sent per second. The 10% packet loss is an artificial packet loss. We simulate packet loss for TCP on client/server with *NetEm* already built into Linux. A client/server with packet loss has the highest latency average about 78267.95 microseconds in contrast to client/server with latency average about 80.14 and PubSub with about 41.31 microseconds. There are also many outliers to be seen in client/server with packet loss, on the top range above the maximum. The outliers at PubSub and client/server are below 450 microseconds other than with packet loss many outliers mostly over 800 microseconds. This is explained by the fact that if a packet is lost in TCP, the packet is sent several times until it arrives at the client, unlike UDP where packets are sent regardless of whether they arrive at the subscriber or not. This means that when a packet is sent again, it costs several hundred milliseconds of latency until the client confirms the arrival of the packet. Excluding the results of TCP and UDP, the open62541 OPC UA PubSub implementation is faster than OPC UA client/server implementation.

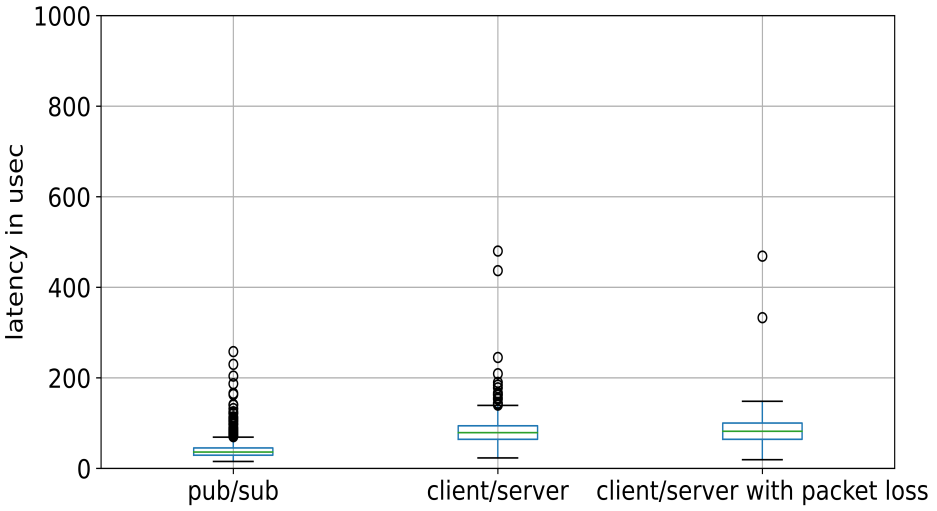

Figure 6.9: OPC UA PubSub and client/server with and without packet loss latency measurements. The plot is limited on the y-axis to 1 milliseconds.

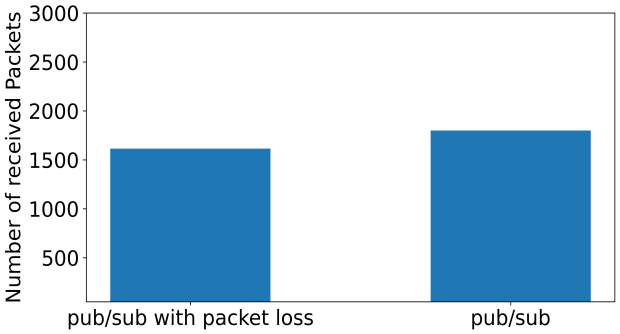

Figure 6.10: Comparison of PubSub with and without packet loss.

Figure 6.9 shows the difference between PubSub without packet loss sends a packet every second for half an hour with PubSub with 10% packet loss with the same procedure as the previous one. There are 1799 packets sent. With PubSub without packet loss 1799 packets are sent and received with pub sub with packet loss many packets are lost, namely 1618 packets. Since UDP packets are only sent without making sure whether the packets have arrived or not. The lost packets are not sent again as with TCP.

6.5 Discussion of Measurements

In this chapter we evaluated several aspects in this work. First, we evaluated the testbed, which is about comparing both OPC UA communication modes client/server and publisher/subscriber. We first performed the performance measurements (RAM, CPU) and then the network latency measurements to show which of the two communication modes consumes less memory and CPU and which of them more suitable for limited-resource devices so that developers can size hardware environments to minimize costs while providing an appropriate performance. We tried to send as many packets as possible to be able to utilize the memory and CPU to the maximum. Within one millisecond, three packets were sent in publisher/subscriber mode and in client/server mode. Developers with more powerful hardware could achieve even higher server or publisher throughputs, while developers with much less powerful hardware could run into bottlenecks. Our results on memory and CPU utilization between client/server and publisher/subscriber have shown that CPU and memory utilization is higher for client/server communication than for PubSub communication. A client has the highest CPU utilization compared to server, publisher and subscriber. In the client/server communication, the server and client CPU and memory utilization were always higher than for publisher and subscriber in PubSub communication mode with the same packet rate.

We have also measured the network latency for client/server and publisher/subscriber. The content of the network message is the timestamp sent on the server or publisher side and captured and evaluated on the client or subscriber side. In case of OPC UA client/server pattern with TCP, we measure the latency between server and client using TCP. In case of OPC UA publish/subscribe pattern with UADP, we measure the latency between subscriber and publisher using UDP as with OPC UA binary data encoding. We simulate packet loss for TCP on client/server with *NetEm* already built into Linux. A client/server with packet loss has the highest latency average. The latency to exchange a message using the client/server pattern is quite high compared to the time required to send the same message using one of the publish/subscribe channels. We can conclude that the time required to receive the published message is higher in client/server mode than in publisher/subscriber mode.

In the second testbed we implemented keyless entry use case with OPC UA PubSub brokerless. We investigate CPU & memory consumption of keyless entry components. Our previous results in the first testbed showed that PubSub is faster than client/server and more energy efficient, so we implemented the keyless entry use case with PubSub. In the application some component are publisher, subscriber or publisher and subscriber at the same time, so the CPU and memory usage differs from each component. *DoorModules*, which was only implemented as a subscriber, consumes at least CPU and memory. *DoorModules* has the lowest memory usage of all other components because it does not contain *WriterGroups* and only subscribes. What also plays a role in memory usage is the message size which also depends on the data types of the messages. Different size messages are exchanged between the keyless entry components depending on the purpose. These results are important to show what hardware the keyless entry implementation can run on and

whether the implementation can run on low performance hardware. Most previous work with OPC UA PubSub have built their testbed only on *Raspberry Pi* with Linux. In general for our implementation of the keyless entry use case there is no specific prerequisite for the device to run it. The only thing we need is network connection with other keyless entry components and some memory and CPU utilization. For purpose of our implementation of keyless entry, it was tried to get the most suitable implementation of the OPC UA stack. We selected the latest version of the open source of OPC UA with a PubSub extension. It is called open62541 stack. It implements the latest version of OPC UA with a PubSub extension. However, the brokerless mode of OPC UA PubSub is still under development.

Our application in OPC UA supports wake-up via network, in which it is possible for clients to wake-up, send a status message to the server and go back to sleep without waiting for a response. The OPC UA PubSub we used is a real-time technology, but it's not time-aware. With addition of latest OPC UA PubSub, *Time-Sensitive Networking (TSN)* will be able to carry different types of messages in real-time, while keeping up the performance of our keyless entry implementation.

In this chapter, the last research questions from **RQ 6.1** to **RQ 6.3** are answered. In this thesis all research questions are addressed and all goals are fulfilled. Based on these results, our keyless application can run with OPC UA PubSub within vehicles.

7. Conclusion

In today's vehicles, up to 80 *Electronic Control Units (ECUs)* communicate over up to six networking technologies. Automotive systems evolved into complex systems containing a reasonable number of distributed ECUs as well as additional sensors and actuators. In-vehicle communication makes special demands on the communication middleware. Onboard communication networks with high bandwidth are also needed to ensure the rapid exchange of necessary information between individual systems. In the first decade of the new century, various Ethernet-based protocols were expected to unify communication, but the effort was only partially effective. There is currently no IP-based middleware that meets the criteria for in-vehicle communication. The implementation of *Service-Oriented Communication (SOC)* protocols into the vehicle's communication system will address these upcoming challenges. *Scalable service-Oriented MiddlewarE over IP (SOME/IP)* is the first standard for IP-based middleware solutions. While SOME/IP is regarded as a promising communication middleware, it lacks security features and does not incorporate security measures such as authentication, integrity, and confidentiality, leaving applications and messages transmitted across the network fully vulnerable to malicious attacks.

This thesis investigates the applicability of an Industry 4.0 protocol for in-vehicle communication by modeling a complex communication scenario in *Open Platform Communications Unified Architecture (OPC UA)* and developing a prototype application for further evaluation. We compared SOME/IP, OPC UA and other *Message Oriented Middleware (MOM)* protocols supporting *publish/subscribe (PubSub)* on the theoretical level such as licensing, real-time, *Quality of Service (QoS)* and security. Then we compared basic features of the most important middleware protocols that support PubSub communication mode. We compared OPC UA and SOME/IP more extensively than the other MOM protocols, since SOME/IP was used in-vehicle communication and the others only in IoT. Furthermore, we have implemented two testbeds. The first was used to evaluate the OPC UA performance using two different communication channels client/server subscription using TCP and publish/subscribe using UDP multicast (UADP). The second one representing in-vehicle communication use case keyless entry. For that, we used an open62541 library, which implements OPC UA PubSub and client/server. We want to proof that the implementation of OPC UA over in-vehicle communication network is possible by using OPC UA. A vehicular communication use case is represented, the requirement and the entities for communication within the use case are specified. Furthermore, we created an information model provided by the OPC UA specifications to describe the meaning of keyless entry instances and the representation of exchanged data.

The contributions of this thesis can be summarized in three parts. A first contribution of this work is the definition and modeling of the keyless entry use case. As second, we designed an information model, how the keyless entry can be structured in semantic data. As third, we performed evaluation and implemented prototypes whereby several performance measurements can be performed. In addition, we investigated the feasibility of the keyless entry use case with OPC UA.

Our research has revealed, that OPC UA is suitable for the chosen keyless entry use case, which represents one of the most complex functionalities with several involved entities. SOME/IP does not support real-time communication and does not include any security functionality or integrate any security measures. In comparison to the other MOM protocols, OPC UA is the only protocol that supports semantic data, which make OPC UA a device centric protocol that focuses on device interoperability, where devices may be used in different systems. In addition, this feature makes it well suited for IoT applications and in-vehicle communication. Furthermore, we made a comparison between client/server and PubSub in terms of resource requirements and consumption (CPU, RAM). In the client/server communication, the server and client CPU and memory utilization were higher than for publisher and subscriber in PubSub communication mode with the same packet rate. Moreover, we investigated the impact of sizing UDP packet overhead of OPC UA publisher on the subscriber performance. The evaluation showed that OPC UA has a small overhead (approximately 30 bytes) when sending out data messages. For each additional *Dataset*, an overhead of 8 bytes plus one byte is added. We can also consider that the amount of *Datasets* affects the CPU and memory consumption in the keyless entry testbed. OPC UA supports the wake-up via network. This reduces the communication overhead to increase the battery lifetime and maximize energy efficiency. With addition of latest OPC UA PubSub with *Time-Sensitive Networking (TSN)*, then OPC UA will be able to support real-time communication and that covers our real-time requirement for keyless entry. Furthermore, OPC UA supports security mechanisms, which include authentication, authorization, confidentiality and integrity.

In this work all research questions are addressed and all goals are fulfilled. Based on these results, our keyless application can run on resource limited devices and maximize energy efficiency by using PubSub with UDP as transport protocol with low message overhead. The use of OPC UA PubSub with UDP for our use case is well suited for TSN networks and devices with constrained resources like sensor nodes and low power processors. OPC UA has its strength in the semantic modeling of information, which makes it a device centric protocol that focuses on device interoperability, where devices may be used in different systems and enables more efficient and autonomous interaction between systems.

In future work, measurements could be made with our keyless entry application with TSN. Furthermore, we did not conduct a comparison of further OPC UA SDKs for PubSub performance, so our results may not be easily generalized to applications using different SDKs. The future work includes also building of separate layers of security for the OPC UA. When this is implemented by all the automobile manufacturers, communication between them becomes way more easier. Also, the future of OPC UA depends on how the industry moves towards. Depending on this direction, the future of this robust, extensible architecture can shape itself.

List of Figures

List of Tables

Listings

Acronyms

ECUs Electronic Control Units

ECU Electronic Control Unit

CAN Controller Area Network

LIN Local Interconnect Network

SOME/IP Scalable service-Oriented MiddlewarE over IP

AUTOSAR AUTomotive Open System ARchitecture

TCP/IP Transmission Control Protocol/Internet Protocol

TCP Transmission Control Protocol

UA Unified Architecture

UDP User Datagram Protocol

OPC UA Open Platform Communications Unified Architecture

SOA service-oriented architecture

LoWPAN Low-Power Wireless Personal Area Networks

BSI Federal Office for Information Security

IEC International Electrotechnical Commission

ICC In-car communication

SOAP Simple Object Access Protocol

HTTP Hyper Text Transfer Protocol

RPC Remote Procedural Call

SD Service Discovery

PubSub publish/subscribe

IP Internet Protocol

TTL Time to Live

SDK Software Development Kit

MOM Message Oriented Middleware

MOST Media Oriented Systems Transport

SOC Service-Oriented Communication

BCM Body control module

LED Light-Emitting Diode

RFID Radio-Frequency Identification

UHF Ultra High Frequency

LF Low Frequency

ID Identification

HF High Frequency

PCB Printed Circuit Board

PDS PublishedDataSet

GPOS General-Purpose Operating Systems

RTOS Real Time Operating Systems

MQTT Message Queuing Telemetry Transport

AMQP Advanced Message Queuing Protocol

lwIP Lightweight IP

OS Operating System

UID User Identifikation

URI Uniform Resource Identifier

MQTT Message Queuing Telemetry Transport

M2M machine-to-machine

OASIS Organization for the Advancement of Structured Information Standards

MQTT-SN MQTT for Sensor Networks

QoS Quality of Service

SCTP Stream Control Transmission Protocol

SASL Simple Authentication and Security Layer

TLS Transport Layer Security

CoAP Constrained Application Protocol

IETF Internet Engineering Task Force

CoRE Constrained RESTful Environments

DTLS Datagram Transport Layer Security

IPSec Internet Protocol Security

DDS Data Distribution Service

OMG Object Management Group

REST Representational State Transfer

XMPP Extensible Messaging and Presence Protocol

SASL Simple Authentication and Security Layer

TLS Transport Layer Security

SSL Secure Sockets Layer

DTLS Datagram Transport Layer Security

ROS Robot Operating System

BSD Berkeley Software Distribution

OSRF Open Source Robotics Foundation

RPC Remote Procedure Calls

ZeroMQ Zero Message Queue

GPL General Public License

PKI Private Key Infrastructure

ACL Access Control Lists

IPC Inter-process Communication

OSEk Open Systems and their Interfaces for the Electronics in Motor Vehicles

TSN Time-Sensitive Networking

UADP UA Datagram Protocol

IoT Internet of things

IBM International Business Machines Corporation

MAC Media-Access-Control-Address

Bibliography

[1] N. Navet, Y. Song, F. Simonot-Lion, and C. Wilwert, "Trends in automotive communication systems," *Proceedings of the IEEE*, vol. 93, no. 6, pp. 1204–1223, 2005.

[2] I. Studnia, V. Nicomette, E. Alata, Y. Deswarte, M. Kaaniche, and Y. Laarouchi, "Survey on security threats and protection mechanisms in embedded automotive networks," in *2013 43rd Annual IEEE/IFIP Conference on Dependable Systems and Networks Workshop (DSN-W)*, pp. 1–12, IEEE, 2013.

[3] C. Corbett, T. Basic, T. Lukaseder, and F. Kargl, "A testing framework architecture for automotive intrusion detection systems," *Automotive-Safety & Security 2017-Sicherheit und Zuverlässigkeit für automobile Informationstechnik*, 2017.

[4] S. Moradpour Chahaki, "On-Board Diagnostics over Ethernet," 2012.

[5] P. Drahoš, E. Kučera, O. Haffner, and I. Klimo, "Trends in industrial communication and opc ua," in *2018 Cybernetics Informatics (K I)*, pp. 1–5, 2018.

[6] I. H. Krüger, E. C. Nelson, and K. V. Prasad, "Service-based software development for automotive applications," tech. rep., SAE Technical Paper, 2004.

[7] A. Zeeb, "Plug and play solution for Autosar software components," *ATZelektronik worldwide*, vol. 7, no. 1, pp. 16–21, 2012.

[8] J. Liu, S. Zhang, W. Sun, and Y. Shi, "In-vehicle network attacks and countermeasures: Challenges and future directions," *IEEE Network*, vol. 31, no. 5, pp. 50–58, 2017.

[9] L. L. Bello, "The case for ethernet in automotive communications," *ACM SIGBED Review*, vol. 8, no. 4, pp. 7–15, 2011.

[10] G. Gopu, K. Kavitha, and J. Joy, "Service oriented architecture based connectivity of automotive ECUs," in *2016 International Conference on Circuit, Power and Computing Technologies (ICCPCT)*, pp. 1–4, IEEE, 2016.

[11] Y. Zhang, W. Ding, Z. Yang, and Q. Zhou, "Research on the key technology and performance analysis of some/ip," in *Proceedings of the 2020 4th International Conference on Electronic Information Technology and Computer Engineering*, pp. 216–221, 2020.

[12] M. Wagner, S. Schildt, and M. Poehnl, "Service-oriented communication for controller area networks," in *2016 IEEE 84th Vehicular Technology Conference (VTC-Fall)*, pp. 1–5, 2016.

[13] AUTOSAR, "SOME/IP Protocol Specification." `https://www.autosar.org/fileadmin/user_upload/standards/foundation/1-0/AUTOSAR_PRS_SOMEIPProtocol.pdf`. year = "2016", Accessed: "2020-08-13".

[14] D. L. Völker, "Scalable service-Oriented MiddlewarE over IP (SOME/IP) overview." `http://some-ip.com/`. year = "2012", Accessed: 2020-08-05.

[15] V. I. GmbH, "Einführung in Automotive Ethernet." https://elearning.vector.com/index.php?&wbt_ls_seite_id=1536276&root=3. year = "2010 - 2020", Accessed: "2020-08-12".

[16] Gururaja, "SOMEI/P." https://at.projects.genivi.org/wiki/download/attachments/16024763/SOME-IP%20Intro.pdf?version=1&modificationDate=1519797378000&api=v2. [Online; posted January-2020, accessed 2020-08-12].

[17] R. Khondoker, B. Reuther, D. Schwerdel, A. Siddiqui, and P. Müller, "Describing and selecting communication services in a service oriented network architecture," in *2010 ITU-T Kaleidoscope: Beyond the Internet?-Innovations for Future Networks and Services*, pp. 1–8, IEEE, 2010.

[18] W. Mahnke, S.-H. Leitner, and M. Damm, *OPC unified architecture*. Springer Science & Business Media, 2009.

[19] J. Pfrommer, A. Ebner, S. Ravikumar, and B. Karunakaran, "Open Source OPC UA PubSub Over TSN for Realtime Industrial Communicatio," in *2018 IEEE 23rd International Conference on Emerging Technologies and Factory Automation (ETFA)*, vol. 1, pp. 1087–1090, 2018.

[20] P. Drahoš, E. Kučera, O. Haffner, and I. Klimo, "Trends in industrial communication and OPC UA," in *2018 Cybernetics & Informatics (K&I)*, pp. 1–5, IEEE, 2018.

[21] J. Pfrommer, A. Ebner, S. Ravikumar, and B. Karunakaran, "Open source opc ua pubsub over tsn for realtime industrial communication," in *2018 IEEE 23rd International Conference on Emerging Technologies and Factory Automation (ETFA)*, vol. 1, pp. 1087–1090, IEEE, 2018.

[22] U. Automation, "Ansi c based opc ua client/server sdk: Introduction to opc ua." https://documentation.unified-automation.com/uasdkc/1.8.0/html/L2OpcUaOverview.html. [Online; posted 2019, updated 2021, accessed 2021-04-13].

[23] T. Pfeiffer, "Realisierung eines verteilten IT-Systems auf Basis von OPC UA," Master's thesis, 2017.

[24] O. FOUNDATION, "Part 1 - Overview and Concepts (Version 1.03)." https://opcfoundation.org/developer-tools/specifications-unified-architecture/part-1-overview-and-concepts/. year = "2017-11-22", Accessed: 2020-08-06.

[25] OPCFoundation, "Opc 10000-14 unified architecture part 14 pub sub | opc ua online reference." https://reference.opcfoundation.org/v104/Core/docs/Part14/. [Online; posted 2020, updated 2020, accessed 2020-12-31].

[26] A. Eckhardt, S. Müller, and L. Leurs, "An evaluation of the applicability of opc ua publish subscribe on factory automation use cases," in *2018 IEEE 23rd International Conference on Emerging Technologies and Factory Automation (ETFA)*, vol. 1, pp. 1071–1074, IEEE, 2018.

[27] F. Pauker, S. Wolny, S. M. Fallah, and M. Wimmer, "Uml2opc-uatransforming uml class diagrams to opc ua information models," *Procedia CIRP*, vol. 67, pp. 128 – 133, 2018. 11th CIRP Conference on Intelligent Computation in Manufacturing Engineering, 19-21 July 2017, Gulf of Naples, Italy.

[28] S. Grüner, J. Pfrommer, and F. Palm, "A restful extension of OPC UA," in *2015 IEEE World Conference on Factory Communication Systems (WFCS)*, pp. 1–4, IEEE, 2015.

[29] J. Troci, ""Model for Evaluation of OPC-UA for Industry-4.0-compliant Communication in Wireless Sensor Networks," Master Thesis (Masterarbeit), Technische Universität Ilmenau, Department of Electrical Engineering and Information Technology," Master's thesis, January 2019.

[30] O. Foundation, "Free OPC UA Modeler." `https://github.com/FreeOpcUa/opcua-modeler`. [Online; posted 2016, updated 2020, accessed 2020-12-07].

[31] Beeond, "UMX Pro – UA Model eXcelerator Professional." `https://beeond.net/umxpro/`. [Online; posted 2005, updated 2020, accessed 2020-12-08].

[32] OPCFoundation, "UA-ModelCompiler." `https://github.com/OPCFoundation/UA-ModelCompiler`. [Online; posted 2005, updated 2015, accessed 2020-12-08].

[33] T. Nolte, H. Hansson, and L. L. Bello, "Automotive communications-past, current and future," in *2005 IEEE Conference on Emerging Technologies and Factory Automation*, vol. 1, pp. 8 pp.–992, 2005.

[34] H.-T. Lim, L. Völker, and D. Herrscher, "Challenges in a future IP/Ethernet-based in-car network for real-time applications," in *Proceedings of the 48th Design Automation Conference*, pp. 7–12, 2011.

[35] N. Navet, Y. Song, F. Simonot-Lion, and C. Wilwert, "Trends in Automotive Communication Systems," *Proceedings of the IEEE*, vol. 93, no. 6, pp. 1204–1223, 2005.

[36] N. Navet and F. Simonot-Lion, "In-vehicle communication networks-a historical perspective and review," tech. rep., University of Luxembourg, 2013.

[37] A. Bose, "Property-based testing: evaluating its applicability and effectiveness for AUTOSAR basic software," Master's thesis, 2020.

[38] S. Fürst and M. Bechter, "AUTOSAR for connected and autonomous vehicles: The AUTOSAR adaptive platform," in *2016 46th annual IEEE/IFIP international conference on Dependable Systems and Networks Workshop (DSN-W)*, pp. 215–217, IEEE, 2016.

[39] L. Slansky and T. Scharnhorst, "AUTOSAR for intelligent vehicles," 2018.

[40] M. Massoud, "Evaluation of an Adaptive AUTOSAR System in Context of Functional Safety Environments," 2017.

[41] T. Gehrmann and P. Duplys, "Intrusion detection for some/ip: Challenges and opportunities," *2020 23rd Euromicro Conference on Digital System Design (DSD)*, pp. 583–587, 2020.

[42] S. Fürst and M. Bechter, "Autosar for connected and autonomous vehicles: The autosar adaptive platform," *2016 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshop (DSN-W)*, pp. 215–217, 2016.

[43] J. Seyler, N. Navet, and L. Fejoz, "Insights on the configuration and performances of SOME/IP service discovery," *SAE International Journal of Passenger Cars-Electronic and Electrical Systems*, vol. 8, no. 2015-01-0197, pp. 124–129, 2015.

[44] J. R. Seyler, T. Streichert, M. Glaß, N. Navet, and J. Teich, "Formal analysis of the startup delay of SOME/IP service discovery," in *2015 Design, Automation Test in Europe Conference Exhibition (DATE)*, pp. 49–54, 2015.

[45] AUTOSAR, "Specification on some/ip transport protocol." `https://www.autosar.org/fileadmin/user_upload/standards/classic/19-11/AUTOSAR_SWS_SOMEIPTransportProtocol.pdf`. [Online; posted 2016, updated 2019, accessed 2021-03-04].

[46] K. Matheus and T. Königseder, *Automotive Ethernet*. USA: Cambridge University Press, 1st ed., 2015.

[47] M. Iorio, M. Reineri, F. Risso, R. Sisto, and F. Valenza, "Securing some/ip for in-vehicle service protection," *IEEE Transactions on Vehicular Technology*, vol. 69, no. 11, pp. 13450–13466, 2020.

[48] T. Gehrmann and P. Duplys, "Intrusion detection for some/ip: Challenges and opportunities," in *2020 23rd Euromicro Conference on Digital System Design (DSD)*, pp. 583–587, IEEE, 2020.

[49] J. Luzuriaga, J.-C. Cano, C. Calafate, P. Manzoni, M. Perez, and P. Boronat, "Handling mobility in iot applications using the mqtt protocol," pp. 245–250, 09 2015.

[50] A. Banks, "MQTT Version 5.0." `https://docs.oasis-open.org/mqtt/mqtt/v5.0/os/mqtt-v5.0-os.html`. [Online; posted 2019, updated 2019, accessed 2021-03-04].

[51] N. Naik, "Choice of effective messaging protocols for iot systems: Mqtt, coap, amqp and http," in *2017 IEEE international systems engineering symposium (ISSE)*, pp. 1–7, IEEE, 2017.

[52] A. Foster, "Messaging technologies for the industrial internet and the internet of things," *PrismTech Whitepaper*, vol. 21, 2015.

[53] O. Standard, "Oasis advanced message queuing protocol (amqp) version 1.0," *International Journal of Aerospace Engineering Hindawi www. hindawi. com*, vol. 2018, 2012.

[54] D. Thangavel, X. Ma, A. Valera, H.-X. Tan, and C. K.-Y. Tan, "Performance evaluation of mqtt and coap via a common middleware," in *2014 IEEE ninth international conference on intelligent sensors, sensor networks and information processing (ISSNIP)*, pp. 1–6, IEEE, 2014.

[55] S. Al-Sarawi, M. Anbar, K. Alieyan, and M. Alzubaidi, "Internet of things (iot) communication protocols," in *2017 8th International conference on information technology (ICIT)*, pp. 685–690, IEEE, 2017.

[56] A. Al-Fuqaha, M. Guizani, M. Mohammadi, M. Aledhari, and M. Ayyash, "Internet of things: A survey on enabling technologies, protocols, and applications," *IEEE communications surveys & tutorials*, vol. 17, no. 4, pp. 2347–2376, 2015.

[57] V. Karagiannis, P. Chatzimisios, F. Vazquez-Gallego, and J. Alonso-Zarate, "A survey on application layer protocols for the internet of things," *Transaction on IoT and Cloud computing*, vol. 3, no. 1, pp. 11–17, 2015.

[58] F. Standard, "1037c: Telecommunications: Glossary of telecommunication terms. national communication system. technology and standards division. washington, dc: General services administration," *Information Technology Service*, 1996.

[59] F. M. Noori, D. Portugal, R. P. Rocha, and M. S. Couceiro, "On 3d simulators for multi-robot systems in ros: Morse or gazebo?," in *2017 IEEE International Symposium on Safety, Security and Rescue Robotics (SSRR)*, pp. 19–24, 2017.

[60] H. Wei, Z. Shao, Z. Huang, R. Chen, Y. Guan, J. Tan, and Z. Shao, "Rt-ros: A real-time ros architecture on multi-core processors," *Future Generation Computer Systems*, vol. 56, pp. 171–178, 2016.

[61] T. ZeroMQ, "Zeromq." `https://www.autosar.org/fileadmin/user_upload/standards/classic/19-11/AUTOSAR_SWS_SOMEIPTransportProtocol.pdf`. [Online; posted 2021, updated 2021, accessed 2021-03-10].

[62] P. Sommer, F. Schellroth, M. Fischer, and J. Schlechtendahl, "Message-oriented middleware for industrial production systems," in *2018 IEEE 14th International Conference on Automation Science and Engineering (CASE)*, pp. 1217–1223, IEEE, 2018.

[63] HELLA, "Keyless Entry system." https://www.hella.com/techworld/uk/Technical/Car-electronics-and-electrics/Keyless-Go-3195/. [Online; posted 2018, updated 2020, accessed 2020-12-17].

[64] S. Rizvi, J. Imler, L. Ritchey, and M. Tokar, "Securing PKES against Relay Attacks using Coordinate Tracing and Multi-Factor Authentication," in *2019 53rd Annual Conference on Information Sciences and Systems (CISS)*, pp. 1–6, 2019.

[65] Intedis, "Keyless Entry Timing Chart." https://www.intedis.com/. [Online; posted 2020, updated 2020, accessed 2020-12-18].

[66] O. Foundation, "Part 5: Information Model." https://opcfoundation.org/developer-tools/specifications-unified-architecture/part-5-information-model/. [Online; posted 2013, updated 2017, accessed 2020-12-14].

[67] S. Profanter, "OPC UA Address Space Explained." https://opcua.rocks/address-space/. [Online; posted 2019, updated 2019, accessed 2020-12-14].

[68] S. Profanter, "From modelling to execution OPC UA Information Model Tutorial." https://opcua.rocks/from-modelling-to-execution-opc-ua-information-model-tutorial/. [Online; posted 2020, updated 2020, accessed 2020-12-14].

[69] S. P. O. U. rocks, "Step 9: Using a custom UANodeSet with open62541." https://opcua.rocks/step-9-using-a-custom-uanodeset-with-open62541/. [Online; posted 2020, updated 2020, accessed 2020-12-10].

[70] open62541, "XML Nodeset Compiler." https://open62541.org/doc/current/nodeset_compiler.html. [Online; posted 2016, updated 2020, accessed 2020-12-11].

[71] J. Kroll, "open62541: Zertifizierter Server auf Basis des Open Source OPC-UA-Stack – Automation/Industrie 4.0 & IoT – Elektroniknet." https://www.elektroniknet.de/automation/industrie-40-iot/zertifizierter-server-auf-basis-des-open-source-opc-ua-stack.169619.html. [Online; posted 2019, updated 2019, accessed 2020-12-30].

[72] A. Burger, H. Koziolek, J. Rückert, M. Platenius-Mohr, and G. Stomberg, "Bottleneck identification and performance modeling of OPC UA communication models," in *Proceedings of the 2019 ACM/SPEC International Conference on Performance Engineering*, pp. 231–242, 2019.

[73] NetTimeLogic, "NetTimeLogic GmbH — OPC UA server on a FPGA using open62541." https://nettimelogic.tumblr.com/post/187371127220/opc-ua-server-with-open62541-running-with-a. [Online; posted 2019, updated 2019, accessed 2020-12-30].

[74] A. Ioana and A. Korodi, "Improving OPC UA Publish-Subscribe Mechanism over UDP with Synchronization Algorithm and Multithreading Broker Application," *Sensors*, vol. 20, no. 19, p. 5591, 2020.

[75] S. Profanter, A. Tekat, K. Dorofeev, M. Rickert, and A. Knoll, "OPC UA versus ROS, DDS, and MQTT: performance evaluation of industry 4.0 protocols," in *Proceedings of the IEEE International Conference on Industrial Technology (ICIT)*, 2019.

[76] O. Foundation, "OPC Unified Architecture Amendment 6 UADP Header Layouts." https://reference.opcfoundation.org/src/v104/Core/docs/Amendment6/readme.htm. [Online; posted 2019, updated 2019, accessed 2021-01-05].

[77] A. Eckhardt and S. Müller, "Analysis of the Round Trip Time of OPC UA and TSN based Peer-to-Peer Communication," in *2019 24th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*, pp. 161–167, IEEE, 2019.

[78] O. Foundation, "OPC Unified ArchitecturePart 6 :Mappings." `http://read.pudn.com/downloads570/ebook/2341940/OPC%20UA%20Part%206%20-%20Mappings%201.00%20Specification.pdf`. [Online; posted 2009, accessed 2021-01-05].

[79] H. Haskamp, M. Meyer, R. Möllmann, F. Orth, and A. W. Colombo, "Benchmarking of existing OPC UA implementations for Industrie 4.0-compliant digitalization solutions," in *2017 IEEE 15th International Conference on Industrial Informatics (INDIN)*, pp. 589–594, 2017.

Appendix

7.1 Creating Writer Groups in OPC UA Publish/Subscribe

Listing 7.1: Add WriterGroup to publisher.

```
1  addWriterGroup(UA_Duration publishInterval, UA_NodeId &
       connectionIdent, UA_Int16 writerGroupId) {
2      UA_NodeId writerGroupIdent;
3      UA_WriterGroupConfig writerGroupConfig;
4      memset(&writerGroupConfig, 0, sizeof(
           UA_WriterGroupConfig));
5      char name[] = "Demo WriterGroup";
6      writerGroupConfig.name = UA_STRING(name);
7      writerGroupConfig.publishingInterval = publishInterval;
8      writerGroupConfig.enabled = UA_FALSE;
9      writerGroupConfig.writerGroupId = writerGroupId;
10     writerGroupConfig.encodingMimeType =
           UA_PUBSUB_ENCODING_UADP;
11     writerGroupConfig.messageSettings.encoding=
           UA_EXTENSIONOBJECT_DECODED;
12     writerGroupConfig.messageSettings.content.decoded.type
13     &UA_TYPES[UA_TYPES_UADPWRITERGROUPMESSAGEDATATYPE];
14     UA_UadpWriterGroupMessageDataType *writerGroupMessage=
15      UA_UadpWriterGroupMessageDataType_new();
16     writerGroupMessage->networkMessageContentMask=
17      (UA_UadpNetworkMessageContentMask)
18      (UA_UADPNETWORKMESSAGECONTENTMASK_PUBLISHERID |
19      (UA_UadpNetworkMessageContentMask)
20          UA_UADPNETWORKMESSAGECONTENTMASK_GROUPHEADER |
21     (UA_UadpNetworkMessageContentMask)
22     UA_UADPNETWORKMESSAGECONTENTMASK_WRITERGROUPID
                                                  (
           UA_UadpNetworkMessageContentMask)
23     UA_UADPNETWORKMESSAGECONTENTMASK_PAYLOADHEADER);
24     writerGroupConfig.messageSettings.content.decoded.data =
           writerGroupMessage;
```

```
25      UA_Server_addWriterGroup(server, connectionIdent, &
            writerGroupConfig, &writerGroupIdent);
26      UA_Server_setWriterGroupOperational(server,
            writerGroupIdent);
27      UA_UadpWriterGroupMessageDataType_delete(
            writerGroupMessage);
28      return writerGroupIdent;
29  }
```

7.2 Different Attributes by a Node in Information Model

Attribute	ID	Description
NodeID	1	Server-unique identifier for the node. Is composed of a namespace index and a node name.
NodeClass	2	Integer enumeration specifying what nodeclass the node belongs to.
BrowseName	3	Used to identify the node when browsing, and is composed of a namespace index and nonlocalized string. Is of the data type QualifiedName.
DisplayName	4	Name to be displayed in a user interface. Is of the data type LocalizedText.
Description	5	Textual description to be displayed in a user interface. Also of the data type LocalizedText.
WriteMask	6	Specifies which attributes of the node can be written to by an OPC UA client. Is of the data type UInt32.
UserWriteMask	7	Specifies which attributes of the node can be written to by the OPC UA client is currently connected to the server. The returned value of this attribute will therefore depend on the currently connected client. Is of the data type UInt32.
IsAbstract	8	Boolean-value that specifies whether the type-node can be used directly by instances, or if it can only be used as supertype to other type-nodes in a type-hierarchy.
Symmetric	9	Boolean-value that specifies whether or not the semantic of a reference is the same in the inverse direction.
InverseName	10	LocalizedText-value that specifies the semantic of the reference in the inverse direction if the reference is not symmetric.
ContainsNoLoops	11	Boolean-value that indicates if the nodes in a view spans a hierarchy without loops or not, when following hierarchical references.
EventNotifier	12	Bit-mask that specifies whether the node can be used to subscribe to events, and whether the event history is accessible/changeable.
Value	13	The actual variable value of a variable-node. The type of value stored under this attribute is specified by the DataType, ValueRank and ArrayDimensions attributes.
DataType	14	NodeId of the DataType-node that defines the data type of the value attribute of the current Variable-node.
ValueRank	15	Int32-value specifying the dimensions of a variable-value array.
ArrayDimensions	16	Optional UInt32-value that specifies the size of each array dimension.
AccessLevel	17	Bit-mask indicating whether the current value of the value attribute is readable and writable as well as whether the history of the value is readable and changeable.
serAccessLevel	18	Same as access level, but with user access rights taken into account.

Minimum Sampling Interval	19	Optional attribute that specify how fast the OPC UA server can detect changes of the value attribute. (This is relevant when the values are not directly managed by the server, e.g., when the node correspond to a third-party sensor that the server is polling.)
Historizing	20	Boolean value specifying whether the server stores history for the variable value.
Executable	21	Boolean value specifying if the method can be invoked.
UserExecutable	22	Same as executable, but with user access rights taken into account.

Table 7.1: The list of all possible attributes the various node classes may contain, listed with their corresponding attribute indexes [78].

7.3 OPC UA Libraries and built-in Data Types

Name	**FreeOpcUa Python.**	**node-opcua**	**OPC UA.NET**	**OPC UA4j**	**OPC UA ANSIC**	**OPC UA Client**	**open62541**	**OPyCua**	**UAF**
Version	0.90.2	0.0.61	1.03.341	0.9.5	1.03.340	1.5.2	0.2-rc2	prealpha	-
Actuality	03.2017	02.2017	02.2017	03.2015	03.2017	02.2017	12.2016	06.2012	09.2016
Developer	Community	Community	OPC Foundation	University	OPC Foundation	Community	Research Institute	Private Person	University
Language	Python	JavaScript	C#	Java	C	C#	C99	Python	C++, Python
License	LGPL	MIT	RCL,GPL 2.0, MIT	CC 3.0 BY-SA	RCL, GPL 2.0, MIT	MIT	MPLv2.0	GPL v3	GNU
Certificate	No	No	Yes	No	No	No	No	No	No
Documentation	much	much	few	very few	very few	very few	very much	few	few
Operating system	Windows, Linux	Windows, Linux, MacOS	Windows	Windows, Linux	Windows, Linux	Windows	Windows, Linux, QNX, Android	Windows, Linux	Windows, Linux

Table 7.2: Free OPC UA implementations [79].

Name	OpenOpcUA	Matrikon OPC UA SDK	proSys	Softing OPC Devel-opment Tool C++	Softing OPC Devel-opment Tool .NET	UA.NET OPC UA SDK	UA ANSI C OPC UA SDK	UA C++ OPC UA SDK	UA High Perfor. OPC UA SDK2
Version	1.0.4.4	-	2.0	5.52.0	1.40.0	2.5.3	1.7.0	1.5.4	1.0.0
Actuality	10.2016	03.2017	11.16	09.2016	09.2016	12.2016	07.2016	12.2016	10.2016
Developer	Company	Company	Company	Company	Company	Company	Company	Company	Company
Language	C, C++	C++	Java, C, C++, .Net	C++	C#	C#	ANSI C	C++	C99
Demo	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Certificate	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	No
Type	Client + Serve	Client + Server	Client + Server	Client + Server	Client + Server	Client + Server	Client + Server	Client + Server	Server
Documentation	very much	very much	much	very much	very much	very much	very much	very much	very much
Operating system	Windows, Linux, VxWorks	Windows, Linux	Windows, Linux	Windows, Linux, VxWorks	Windows	Windows	Windows, Linux, VxWorks, QNX, EUROS	Windows, Linux, VxWorks, QNX	Windows, Linux

Table 7.3: Commercial OPC UA implementations [79].

ID	Name	Description
1	Boolean	A two-state logical value (true or false).
2	SByte	An integer value between -128 and 127.
3	Byte	An integer value between 0 and 255.
4	Int16	An integer value between -32768 and 32767.
5	UInt16	An integer value between 0 and 65535.
6	Int32	An integer value between – 2147483648 and 2147483647.
7	UInt32	An integer value between 0 and 4294967295.
8	Int64	An integer value between – 9223372036854775808 and 9223372036854775807.
9	UInt64	An integer value between 0 and 18446744073709551615.
10	Float	An IEEE single precision (32 bit) floating point value.
11	Double	An IEEE double precision (64 bit) floating point value.
12	String	A sequence of Unicode characters.
13	DateTime	An instance in time.
14	Guid	A 16 byte value that can be used as a globally unique identifier.
15	ByteString	A sequence of octets.
16	XmlElement	An XML element.
17	NodeId	An identifier for a node in the address space of an OPC UA server.
18	ExpandedNodeId	A NodeId that allows the namespace URI to be specified instead of an index.
19	StatusCode	A numeric identifier for a error or condition that is associated with a value or an operation.
20	QualifiedName	A name qualified by a namespace.
21	LocalizedText	Human readable text with an optional locale identifier.
22	ExtensionObject	A structure that contains an application specific data type that may not be recognized by the receiver.
23	DataValue	A data value with an associated status code and timestamps.
24	Variant	A union of all of the types specified above.
25	DiagnosticInfo	A structure that contains detailed error and diagnostic information

Table 7.4: OPC UA built-in DataTypes [78].

I declare that I have developed and written the enclosed thesis completely by myself, and have not used sources or means without declaration in the text.

Würzburg, 28. April 2021

.....

(Ramadan Shweiki)