Antti Konttinen

ARCHITECTURE OF INDUSTRIAL DEVICE INTERFACES

Master's thesis
Faculty of Engineering and Natural Sciences
Examiners: Professor Matti Vilkko
Assistant Professor David Hästbacka
October 2021

ABSTRACT

Antti Konttinen: Architecture of Industrial Device Interfaces
Master's thesis
Tampere University
Master's Degree Programme in Automation Engineering
October 2021

Device interfaces are needed when a system needs to communicate with devices, which aren't directly compatible with the system. The interface is used to translate the data passing from the system to the device to make communication possible. Industrial environments have strict requirements for the dependability and performance of the software because real and potentially dangerous machinery and processes are involved. This makes the architecture of an industrial device interface a crucial factor when designing new implementation because otherwise the device interface could malfunction and cause real harm to the environment.

This thesis targets to find a suitable architecture for an industrial device interface by using an architectural pattern as the basis. Integration methods for industrial environments are studied to gather information about the concepts and requirements for the selection of the architecture. Four different architectural patterns (layered, event-driven, microkernel, microservice) are studied and compared to find the best alternative for the architecture. The remaining of the architectural patterns are evaluated to determine if some of their key features could be implemented to the overall architecture.

Performance, maintainability, configurability, safety, and security were determined as the main requirements for the device interface. The chosen architecture uses the microkernel architectural pattern as the baseline. The microkernel architecture uses a core system to implement the necessary functionalities for the software to be operational. Plugin components connect to the core system and offer additional features on top of the core system. In the device interface, the core system handles the communication to the external system and initialization of the plugin components, while the plugins act as device drivers to connect to the devices. Microservice type of architecture was chosen to be supported by allowing multiple core systems to be run parallel with each other.

The designed architecture was tested by implementing a demo interface supporting communication with a Modbus slave device. The results of testing showed that the implementation fulfilled the requirements defined for the device interface rather well. The performance was on a good level and it was easily configurable. Maintainability for the plugin components was good, but if the core system would require changes, it could affect other parts as well. Security was handled well, but safety could become a problem if the core system would fail. Overall, the implementation complied with the decided architecture, even though definite conclusions couldn't be made based on the implementation, because the tests were rather limited.

Keywords: architecture, interface, industrial, pattern, device

TIIVISTELMÄ

Antti Konttinen: Teollisuuden Laiterajapintojen Arkkitehtuuri
Diplomityö
Tampereen yliopisto
Automaatiotekniikan DI-ohjelma
Lokakuu 2021

Laiterajapintoja tarvitaan järjestelmien kommunikoidessa laitteiden kanssa, jotka eivät ole suoraan yhteensopivia käytössä olevien järjestelmien kanssa. Rajapintoja käytetään järjestelmien ja laitteiden välillä liikkuvan tiedon muokkaamiseksi yhteensopivaksi kommunikoinnin mahdollistamiseksi. Teollisuuden ympäristöillä on tiukat vaatimukset sovelluksen käyttövarmuudelle ja tehokkuudelle johtuen niihin liittyvistä mahdollisesti vaarallisia laitteista ja prosesseista. Sen vuoksi teollisuuden laiterajapintojen arkkitehtuuri on tärkeässä roolissa uusia toteutuksia suunniteltaessa, koska muutoin laiterajapinta voisi toimia virheellisesti ja aiheuttaa todellista vahinkoa ympäristölle.

Tämän opinnäytetyön tarkoituksena on löytää teollisuuden laiterajapinnoille soveltuva arkkitehtuuri käyttämällä hyödyksi arkkitehtuurimalleja. Työssä tutkitaan teollisuusympäristöihin liittyviä integrointimenetelmiä sekä käsitteitä, jotta arkkitehtuurin valintaan vaikuttavat vaatimukset voidaan määritellä. Neljä erilaista arkkitehtuurimallia (kerros, tapahtumapohjainen, mikropalvelu sekä mikrokernel) valitaan vertailtavaksi arkkitehtuurille parhaan vaihtoehdon löytämiseksi. Jäljelle jääneiden arkkitehtuurimallien ominaisuuksia arvioidaan, voidaanko miitä käyttää hyödyksi lopullisessa arkkitehtuurissa.

Suorituskyky, ylläpidettävyys, konfiguroitavuus sekä turvallisuus valittiin laiterajapinnan tärkeimmiksi vaatimuksiksi. Lopullinen arkkitehtuuri pohjautuu mikrokernel-arkkitehtuurimalliin. Mikrokernel-arkkitehtuurissa käytetään mikroydintä sovelluksen välttämättömien ominaisuuksien toteuttamiseksi. Ytimen moduulit yhdistyvät mikroytimeen ja tarjoavat ylimääräisiä ominaisuuksia mikroytimen rinnalle. Suunnitellussa laiterajapinnassa mikroydintä käytetään ulkoisten järjestelmien väliseen kommunikointiin sekä ytimen moduulien alustamiseen, jotka toimivat laiteajureina ympäristöistä löytyville laitteille. Mikropalvelun kaltaista arkkitehtuuria tuetaan mahdollistamalla useamman mikroytimen rinnakkainen toiminta.

Lopullinen arkkitehtuuri testattiin toteuttamalla testirajapinta, joka tuki kommunikointia Modbus-orjalaitteen kanssa. Toteutuksen testaus osoitti, että testirajapinta täytti sille asetetut vaatimukset melko hyvin. Suorituskyky oli hyvällä tasolla ja rajapinta oli helposti konfiguroitavissa. Ytimen moduulien ylläpidettävyys oli hyvä, mutta mikroytimeen tehtävät muutokset voivat mahdollisesti vaikuttaa sovelluksen muihin osiin. Tietoturvan käsittelyssä ei ollut ongelmia, mutta toiminnallinen turvallisuus voisi kärsiä, jos mikroydin menisi vikatilaan. Kaiken kaikkiaan toteutus noudatti hyvin suunniteltua arkkitehtuuria, vaikkakaan täysin varmoja johtopäätöksiä ei voitu tehdä toteutuksen pohjalta, koska testitapaukset olivat melko rajalliset.

Avainsanat: arkkitehtuuri, rajapinta, teollinen, malli, laite

PREFACE

I wish to thank Pinja Digital Oy and all the involved personnel within the company for the opportunity to make this thesis work possible. Special thanks to my supervisor Kosti Karppinen for the support and feedback, which helped significantly especially during the initial phase of the thesis work.

I am also grateful to my examiners Professor Matti Vilkko and Assistant Professor David Hästbacka for their advice and guidance regarding the thesis work.

Finally, I wish to express my gratitude to my family and friends for the support they have given me during the writing process, which encouraged me to finish the thesis work in a reasonable time.

Tampere, 19th October 2021

Antti Konttinen

CONTENTS

1. Introduction 1
1.1 Research goals and questions 2
1.2 Structure 3
2. Integration methods of industrial devices 4
2.1 Interface 4
2.2 TCP/IP 5
2.3 Fieldbus 7
2.4 OPC Unified Architecture 9
2.5 Database 10
2.6 Control systems and devices 11
2.7 Related works 12
3. Architectural patterns 14
3.1 General 15
3.2 Layered 16
3.3 Event-driven. 19
3.4 Microkernel 23
3.5 Microservice 26
4. Requirements and selection of the architecture. 31
4.1 Environment. 31
4.2 Technologies 32
4.3 Requirements 33
4.4 Selection and analysis. 34
5. Implementation and observations 44
5.1 Overall architecture 44
5.2 Implementation 46
5.3 Evaluation 50
6. Conclusion 52
References. 54

1. INTRODUCTION

Software architecture describes the structure of a program, relationships between subsystems, and the overall principle of the design. It is used to represent the public side of the software while leaving out the private parts of the software. Software architecture is supposed to meet the requirements of the software and act as a blueprint for the whole system. Delicately engineered and correctly documented architecture makes the development easier during the development, which also applies to maintaining and updating the software after the release.

In general, the problems occurring in software development are recurrent and are most likely solved by somebody else. Solving already resolved problems isn't usually in the interests of companies funding the software development. Architectural patterns are general-purpose solutions for frequently occurring problems existing in software development. Architectural patterns define the core elements in software architecture. [1] The goal of the architectural patterns is to streamline software development if the chosen pattern fulfils the requirements of the software.

Industrial software differs from other kinds of software (e.g., websites or games) by controlling real-time processes and actual machinery. Because of that, the consequences of a software malfunction could be substantial and potentially dangerous when compared to most of the other applications. The software must be dependable and fulfil the real-time requirements of the environment when needed. The possibility of a hardware failure must be considered in the software, which makes the development even more challenging.

A device interface is a software application, which links a physical device with the target software. The target software can communicate with the device by reading data and sending commands via the interface. The reliable and fast operation of the interface ensures that the information read from the devices is sent correctly and in time to the target environment. If architectural patterns were used in the design of the architecture, the selected architectural pattern would have a big impact on the outcome, because an incorrectly chosen pattern can make the interface intolerably slow and use an excessive amount of resources, when compared to the requirements.

1.1 Research goals and questions

The goal of this thesis is to decide the overall architecture for an industrial device interface given the requirements of a specific use case, which includes different environments with different layouts and configurations. The device interface should operate effectively and enable further development for new features. It should be able to connect to different devices and systems found in environments, such as programmable logic controllers and automation systems by using various communication methods, such as TCP/IP and fieldbuses. The devices should operate completely isolated from each other from the device interface's point of view, which means that problems in one device shouldn't affect the other. Demo software of the device interface with support for a single device type will be implemented for testing purposes by using the overall architecture proposed in this thesis. The research questions are following:

- Which architectural pattern of the selected patterns suits best for developing an industrial device interface?
- How could the patterns be combined to improve the overall architecture?
- How successful was the implementation of the device interface using the specified architecture?

Four different architectural patterns are proposed and compared to find a baseline for the architecture. The selected patterns for the evaluation are layered, event-driven, microkernel, and microservice architectural patterns. The patterns were selected by their different properties and each of them was thought of initially as a suitable solution for the problem. There are a lot of different architectural patterns in addition to the selected patterns, which could have been evaluated in this thesis as well. Service-oriented, object-oriented, entity component system and pipe and filter architectural patterns are examples of the patterns, which were left out.

The best architectural pattern for the device interface will be chosen by analysing the selected patterns using literature as the source material. The patterns will be evaluated by different metrics and qualities, and the best overall alternative will be selected as the main architectural pattern. After the selection, the overall architecture will be based on the previously selected pattern, but also the possibility of combining features from other patterns will be evaluated and potentially implemented to the overall architecture. The device interface will be implemented by following the overall architecture for a single device type. Finally, the implementation will be evaluated based on the requirements defined earlier.

1.2 Structure

The industrial devices and their communication methods for field devices, such as field buses, are discussed in chapter 2. Similar works by others are also introduced there. Chapter 3 gives an overview of the four selected architectural patterns, which could be used to design a device interface. The main principles of the patterns are explained along with some examples of use cases for each pattern based on other works. Chapter 4 describes the background and the practical environment for the device interface. Also, the architectural patterns will be compared for the selection of the architecture and the possibility of combining the patterns is examined. The overall architecture and the implementation for the device interface along with the considerations will be presented in chapter 5. The conclusion and an overview of this thesis are given in chapter 6.

2. INTEGRATION METHODS OF INDUSTRIAL DEVICES

This chapter describes the basics of an interface and different communication methods and concepts used in industrial environments. The fundamentals of TCP/IP communications standard are introduced because it is mainly used in the communication methods described later. Fieldbuses are described because they are highly used in industrial environments for communication. Modbus is one of the protocols used in fieldbuses and it is opened up more specifically in this thesis. OPC Unified Architecture is a communication protocol, which is introduced because of its widespread use in the industry. Databases are used for storing and handling lots of data, which could be required in the industrial environments and thus handled in this thesis. The most common field devices in the industry are described to give an idea, which devices the device interface could connect to. Finally, some examples of previous studies for the device interfaces are given.

2.1 Interface

Interface in general is a place where two or multiple objects are brought together to communicate with each other. The objects are supposed to have public access points which the other objects can use to connect themselves to and communicate with each other. Traffic between the objects can be specified in the interfaces; some objects could be only able to either receive or send information, while some could do both through the interface. [2, pp. 33-34]

In computing, the objects communicating via the interface could be either hardware or software. Hardware interfaces are mechanical ports or hubs, which are used to transfer incoming and outgoing signals to the devices they are connected to. Hardware interfaces require software to interpret the signals passing through, which is called a device driver. Device drivers are used to enabling communication between multiple different components of a computer system. The components can be either software or hardware and they can also be combined. The device driver is supposed to translate the messages between the components and send them forward to create a connection. They connect to components using interfaces, which the components provide for communication. [3]

Software interfaces, which are also called application programming interfaces (API), are used to allow communication between two separate software. An API offers a communication gateway by exposing only the necessary parts of the software to the public, principally every part of the software that isn't necessarily required to be public are hidden. It defines the possible calls which can be made to the software from outside and what information is required to be sent in the request. [4]

2.2 TCP/IP

Internet protocol suite, also known as TCP/IP, is a communications standard for connecting different devices, such as computers, mobile devices and industrial equipment, through the internet. It has been a de facto standard for internet communication for 30 years. TCP/IP consists of different protocols and standards, that defines the way data and connection are handled during the communication. [5, p. 34] It is split into five different layers, which define more detailed abstraction levels of the standard and take care of different parts of the communication [5, p. 44].

TCP/IP consists of five different layers, which are physical, data link, network, transport and application. Physical is the lowest layer of the TCP/IP standard, and it is used to define different transmission mediums, such as connectors and cable types used to transmit data. The physical layer also defines the topology and data rate of the network. The data link layer defines the protocols and methods which are needed for connecting to the medium shared between devices in local networks without routers. For example, in wireless Wi-Fi networks, multiple devices may be connected to the same medium. [5, pp. 44-45]. The data link layer adds information about the devices to the headers of the data packets sent over the network, so the packets are sent to the correct devices [5, p. 138]. It also provides error control in case of faulty data frames sent by using checksum bits, so the data can be sent again. Ethernet is an example of a widely used data link protocol [5, pp. 40-45].

Internet layer specifies methods and protocols that are used to send data packets across the internet. It uses individual addressing for sending the packets between the devices, IP address is a well-known addressing scheme used in TCP/IP. Internet layer also provides routing for the packets by sending each packet individually through an optimal route, if multiple routes are available. [5, p. 45] Transport layer is used only by the end hosts to provide communication between networks that are separated by a router. There are a lot of different transport layer protocols available, but the most used protocols are TCP and UDP. The protocols offer different kinds of services and features, which can be chosen according to the needs of the application. [6, p. 9] Finally, the application layer offers protocols for hosts to connect to other hosts either in the same or different network. The application layer relies on the transport layer to handle the connection between hosts, while the actual data traffic is accomplished either by client-server or peer-to-peer methods. [6, pp. 8-9]

Transmission Control Protocol (TCP) is one of the most widely used transport layer protocols in TCP/IP. TCP uses connected-oriented communication, which means that TCP requires an active connection to be established between the applications communicating with each other before the data can be transmitted. The connection is established by a 3-way handshake process, in which three different messages are sent between the applications before an active connection is set up. For TCP to send data between applications, it must convert the data into smaller pieces, which are called chunks. They must be small enough for IP to carry them to the destination, but they don't have to be of a fixed size. A TCP header, which contains information about the connection properties, is added to the chunks, which creates a TCP segment. The segment is then sent to the IP layer to form a packet. TCP protocol excepts an acknowledgement from the destination application for each segment sent and if the acknowledgement doesn't arrive before the timeout, it tries to send the segment again. The headers of the segments contain a checksum, which is used to verify the integrity of the segment in the destination. If the checksum is invalid, the destination won't send an acknowledgement and thus the segment will be resent after the timeout. [5, pp. 753-777]

Internet Protocol (IP) is a protocol used in the internet layer of TCP/IP to deliver the packets to correct destinations. IP is used as a "best-effort" service, so it doesn't check if the packets arrive at the destination or not, that must be done in the upper layers of TCP/IP, for example in the transport layer by TCP. All the individual packets are handled separately and sent through the shortest route possible. It means, that it is possible for a packet sent after the previous one to arrive first if the optimal route changes between the packets. The packets are sent by IP as datagrams, which consists of a header and a payload. The header part includes the IP addresses for the source and the destination, and other relevant data, which is needed for the datagram to be sent and routed correctly. The payload contains the actual data, which is received from the upper layers. [5, pp. 264-272]

TCP/IP protocol uses ports to manage communication between different applications. They act as the endpoints for communication by allowing the applications to connect to each other using the same port. A server could ping a specific port, while the connecting clients would request access to the server through that same port. Ports are used alongside IP addresses to identify the correct destination for the request. Many of the available ports are reserved for common services, such as SSH, which uses port 22 and HTTP, which uses port 80. [5, pp. 54-59] This allows managing a specific type of network traffic by opening the corresponding ports for the services through a firewall. [5, p. 409]

2.3 Fieldbus

Fieldbus is a network, which is mainly used to connect industrial devices and control interfaces. It allows two-way communication in Local Area Network environments while providing high-speed and real-time data transmission. Traditionally the devices were connected to each other before fieldbuses by using a star topology, in which all the devices that needed to communicate between each other, required a cable to be connected between them. Instead of wiring every device requiring data transmission directly to each other, fieldbuses allow the devices to be connected using different topologies than a star, which reduces the number of cables needed. For example, in a ring topology, all the devices have two interfaces, input and output, and the devices are connected one after another like a chain. The topologies also enable the data moving in the fieldbus to be read from anywhere in the bus, which means that every field device has the access to the same information and thus makes the operation of the devices more reliable. [7, pp. 31-51]

Fieldbus protocols operate in digital format. Initially, they used their own communication protocols in a data-link layer of OSI-model to transport the information between devices. From the beginning of the 2000s, Ethernet has started to grow in popularity as an alternative protocol in the data-link layer. The protocols using Ethernet as the data-link layer are called Industrial Ethernet (IE). Ethernet wasn't originally intended to handle real-time communication and that's why it wasn't considered during the early days of the fieldbuses. The fact, that Ethernet is used in the office networks, would make the integration of industrial and office networks seamless when compared to traditional fieldbuses because all the information could be moved via a single network. Initially, the fieldbus protocols were modified to use TCP/IP and send the packets as TCP or UDP, without altering the Ethernet standards. This meant that the strict real-time applications couldn't operate using those kinds of solutions. The solutions for the most demanding requirements required either modifications to the higher layers of the network stacks or completely new communication stacks, which made those protocols incompatible with other protocols. [7, pp. 67-71]

Fieldbus protocols are designed to follow the OSI model, which is used to model the communication properties of networks using 7 different layers. The complete OSI model would be too complicated and heavy to be used by fieldbus protocols because they require fast real-time communication. Instead, they use layers of OSI-model, which are necessary for data transmission but leave out the layers which would complicate the data transmission too much. OSI layers 1, 2, and 7 are typically used in fieldbus protocols. The names for the layers are respectively transport, data-link and application, which are like those found in TCP/IP. Some protocols might use one or more of the remaining layers 3-6, which are called respectively network, transport, session and presentation. [7, pp. 48-50]

There are a lot of different fieldbus standards available, which are created mostly by different organizations. IEC 61158 is a collection of fieldbus standards, which is maintained by International Electrotechnical Commission (IEC). So far 26 different fieldbus standards have been defined in IEC 61158 [8]. The Industrial Ethernet protocols are defined in IEC 61784 standard. The number of standards is relatively high because companies developed their own standards during the 1980s without consulting each other. When the different implementations were finally standardized by IEC, the solutions were too different to be combined. [7, pp. 37-39]

2.3.1 Modbus

Modbus is a fieldbus protocol that was created in 1978 by Modicon Inc (Schneider Electric since 1999). It was created to allow field devices and logic controllers to communicate with each other. The protocol has been popular from the beginning and has been considered as a de facto fieldbus protocol, which has made it the most supported fieldbus protocol amongst industrial devices. Modbus operates at an application layer of the OSI model and is used for client/server communication between automation devices. Modbus client is a device, which makes requests to other devices and Modbus servers are the devices, which respond to the clients and send the information to them. Modbus can use different serial communication protocols for data transfer, but also TCP/IP standard is supported, which makes it compatible with Ethernet-based systems. [7, pp. 383-384]

The data transmission in Modbus protocols is started, when a client sends a request to the server to perform an action, which could be for example a request for information or an action to be executed. The client/server interactions in Modbus protocol can be either broadcast or unicast. In broadcast mode, the response from the server is not expected and the client will just wait for a pre-defined amount of time before it continues normally. In the unicast type of interaction, the client waits for the acknowledgement of the response, which the server is supposed to send in a limited time frame. The client can also send the same request to multiple servers simultaneously and the servers can respond to multiple clients. [7, pp. 389-390]

Modbus application protocol payload is called Modbus protocol data unit (PDU), which consists of code and data fields. Modbus APDU, which is the same across the data-link protocols, adds a unit id to the PDU. Modbus servers are identified by a unit ID, which is sent as a part of the request from the client to address the request to the correct server. The size of the unit ID field in the message is one octet, which equals 8 bits. The code field is used to send encoding information about the request and the size of it is also one octet. The data field contains additional information on the request and the requested data in the response. In case of a failure, an exception code is sent as a response in the data field. The maximum size of the data field is 252 octets. The limit comes from the

RS-485 protocol's limit of 256 octets per frame, which was the physical layer of the first Modbus implementation. The remaining four octets are reserved for a unit id, a code field and two cyclic redundancy check (CRC) octets, which are used for error detection. [7, pp. 387-389]

Modbus protocol supports different data types, from which the most used are discrete, coil, input register and holding register. Discrete is a single bit data type, which is used as inputs and can only be read. The coil can also store single bits, but they can both be read and written by another device. Both input register and holding register can store 16-bit words, the difference being that while the input register can only be read, holding registers can also be written. The PDU data field contains a function code, which has information about the used data type. The function codes also specify along with the data type if the data will be read or written and the client/server interaction type. Function codes can also specify other types of data transmission, for example, a request for diagnostic information. [7, pp. 392-394]

Modbus supports transmission over TCP/IP standards, which is called Modbus TCP. It uses the standard Modbus application protocol in the application layer and Ethernet in the data link layer, TCP and IP are used in transport and network layers of the OSI model. Modbus TCP uses APDU with an additional header part when sending information through TCP/IP stack. [7, pp. 402-410] The header contains a transaction ID, protocol ID and length parameters, which are 2 octets each and a unit identifier parameter, which uses 1 octet [9]. The transaction ID is used to identify a single Modbus request/response transaction, which is used because of the high amount of possible IP addresses and unit IDs of the servers. Protocol ID is a unique identification of the used protocol, which is needed in TCP/IP if the protocol is not identified with a port number. Modbus TCP uses a dedicated port in communication between the devices, the default port being 502, while the protocol ID is set to 0. The length field tells the overall length of the message and is used to verify that the whole message is received correctly. [7, pp. 402-410] The unit identifier is used to identify different slave devices in serial line networks. In Modbus TCP it isn't needed, as the slaves are identified by the IP addresses, so a default value of 255 must be used as the unit identifier. [9]

2.4 OPC Unified Architecture

OPC Unified Architecture (OPC UA) is a communications protocol used in industrial applications, especially in different types of automation systems. It was created to enable platform-independent communication between different devices and applications by offering unified communications methods. The predecessor for OPC UA is OPC Classic, which was designed to be used as a device driver interface standard. OPC UA widened the scope of the previous standard by acting as a system interface to support a wider range of applications. It specifies strict operational requirements, which include reliability, scalability, performance and security, along with other factors it has to take care of. OPC UA was published in 2006 and it is maintained by OPC Foundation. [10]

OPC UA achieves platform-independent communication by data modelling methods and transport mechanisms. Data modelling allows different types of devices and control systems to communicate with each other by standardizing the transmitted messages into specified categories. Despite the vendor of the device, using OPC UA as the communications protocol allows the data to be sent in a format, which is compatible with any client that supports OPC UA. Vendor-specific data is transmitted within the messages to allow more detailed information to be sent during communication. [10]

OPC UA could transfer the messages by using two different encodings, which are OPC UA Binary and XML. OPC UA Binary is the more efficient encoding of the two, because it transfers only a small size of data at a time, which speeds up the encoding and decoding processes. It translates the messages into a binary stream, which is an efficient method to transfer data. XML encoding is used to transfer data into applications, where the representation of the data is more important. [10] XML is a text format, which is widely used across the Internet [11]. Corporate level systems, such as enterprise resource planning (ERP), usually use XML encoding to transfer data [10].

Transport protocols are used to establish a connection between OPC UA devices and applications. OPC UA uses two different transport protocols, which are UA TCP and SOAP/HTTP. UA TCP is a transport protocol, which has been implemented on top of TCP protocol. It allows managing buffer sizes of the transmitted data at the application level, error recovery on the transport level and allows OPC UA servers to share a single IP address and port. SOAP/HTTP is used to transfer data in Web Service environments. It uses HTTP protocol at the transport level, which could simplify managing the network traffic by not needing to open additional ports. SOAP is a network protocol, which handles the data exchange between applications by using XML as the data format. [10]

2.5 Database

Databases are places, where organized information can be stored. For example. databases could be physical storages of collected samples or information stored with pen and paper, but usually, they operate electronically in a computer. The data is typically stored in tables, which have different columns representing the properties of the data. The rows in the tables contain the actual values corresponding to the columns. The data can be accessed from other systems by making queries to the tables on the databases. [12, pp. 33-34]

Databases can be split into two main categories, which are operational databases and analytical databases. Operational databases are used to access and modify the data in real-time. The data is always changing and represents up-to-date information of the database. The operational databases are managed by Operations Database Management Systems, which can add, change or delete data in real-time. Analytical databases on the other hand are used to store historical information, which in general won't be updated after having been saved to the database. Historical data can be used to analyse trends and other metrics, which could help for example in detecting possible issues in the process, from which the data is collected. Analytical databases can be used to store the real-time data from operational databases to be examined later. [12, pp. 33-35]

Relational databases are used to store information, which has relations between different types of data. The relations are stored in tables, in which the distinct data rows are identified by unique values. This ensures the integrity of the data because duplicate rows are not allowed, and every data row will be identified. Because of the unique ids, the order of the data rows isn't relevant, which means that the physical location and the structure of the data don't have to be known when dealing with the data. This makes the databases more robust against changes in both physical and logical functions of the database because the applications using the databases don't have to be aware of those changes. The relationships can be categorized as one-to-one, one-to-many and many-to-many, from which one category is used per a table pairing. The relational databases are maintained and managed in relational database management systems (RDBMS). [12, pp. 41-47]

2.6 Control systems and devices

Control systems are used to control the overall logic and operation of the automation systems. Distributed control systems (DCS) handle the overall logic of the plants by having multiple distributed computers controlling the field devices, while programmable logic controllers (PLC) are used to control individual devices by providing inputs and outputs. [13] Field devices are used in industrial automation to control or measure processes found in the environments. The devices could be different hardware handling the operations in the field, such as actuators and sensors.

DCS could be used to control large portions of the industrial environments' functionalities. It allows reliable and redundant operation by having multiple distributed computer systems controlling the application logic. They are connected by using multiple control loops, which ensures the redundancy of the system if a computer should fail at any time. DCS is controlled through a human-machine interface (HMI), which is a PC having access to the DCS and the data within it. Field controlling station (FCS) is used in DCS to handle the controls in the field. It supports both analogue and digital control by using respective I/O cards. [13]

PLC is used to control field devices by providing inputs and outputs for communication. They could operate independently after configuration, but different fieldbuses are usually supported so that external systems can control the outputs and read inputs from them. Communication between field devices and PLCs is handled by input and output modules, which can be added to the PLCs as needed. The input and output points are separated by a unique address, which is being used by the logic within the PLC or by external systems to control the field devices. PLCs support various programming languages, such as structured text, ladder diagram and instruction list. Modern PLCs use IEC 61131-3 standard as a baseline for the programming language to improve the compatibility of devices made by different vendors and also to make the code more unified. [14]

Field devices managed by control systems in industrial environments consist mainly of different actuators and sensors. Actuators are used to execute mechanical tasks required in the field, such as lifting and rotating machinery. They could be for example pumps, electric motors and hydraulic devices. The control systems communicate and send information to the actuators according to their logic. Sensors are used to measure different values and metrics across industrial environments. Sensors can be divided roughly into two categories based on the types of values they can measure. Discrete sensors measure only binary values of the target object, such as if a gate is open or not. Continuous sensors measure values that aren't limited to two states, but possibly an indefinite range of states, such as temperature or speed. Discrete sensors have usually digital outputs, while continuous sensors could have either digital or analogue output based on the application. [15]

2.7 Related works

There is a lot of content in the literature about different interfaces, which are supposed to link two separate systems together for communication purposes. Cecilio and Furtado suggest their own architecture called MidSN, which allows clients to communicate with embedded devices. The work emphasizes reconfigurability and interoperability for the interface, while also taking into account the performance requirements for industrial environments. The architecture of MidSN consists of networks of devices, which the client applications can connect to. Each device in the network has a node, to which the clients can connect. The nodes offer standard interfaces for every type of device, which allows the clients to make the same calls to every device. The configurability on runtime is achieved by allowing clients and other nodes to send commands directly to the devices. [16]

Interfaces are needed in different Industrial Internet of Things (IoT) applications for communication between devices. Balasubramanian *et. al.* propose in their work an architecture for time-sensitive industrial IoT applications, which uses Software Defined Network

(SDN) for managing resources of the hardware. The work underlines reconfigurability and performance, while also taking scalability into account. The overall architecture consists of a control plane and data plane, which handle different parts of the functionalities. The control plane is used to configure the data plane and also to calculate the optimal routes for packets passing through the interface. The data plane consists of Time-Sensitive Networking (TSN) switches, which make sure that the data is sent forward at the correct time and order. [17]

OPC UA allows data interoperability between devices made by different vendors. Shin proposes in his work an architecture for an interface handling data exchange between OPC UA and predictive model markup language (PMML). The proposed architecture follows a chained server pattern, which incorporates multiple different servers and clients to operate in co-operation. Data is received from different applications by the first server, which transforms it into an OPC UA-compliant format. A client is used to receive the OPC UA data from the first server and to transform it into PMML documents. The next server will convert the data into PMML objects, which are made compatible with OPC UA again. The last client can receive the OPC UA-compliant PMML objects and use them for their own purposes. [18]

Home automation requires interfaces to enable the communication between devices control equipment. Song *et. al.* propose in their work a multi-interface gateway architecture to be used in home automation networks. The architecture is illustrated as a horizontal stack of components, which handle different parts required by the overall design. Different data transmission techniques are supported by having their own interface drivers for each communication method. Data integrity and synchronization is handled by a mutually exclusive (mutex) object, which handles sending the data from interface drivers forward. An embedded operating system is used to send the data from the mutex object to the radio, which can communicate with other objects within the system. The data from the embedded operating system is also shown in a graphical user interface, which is the topmost component in the architecture. [19]

3. ARCHITECTURAL PATTERNS

"The software architecture of a computing system is the set of structures needed to reason about the system, which comprises software elements, relations among them, and properties of both [20, p. 1]." It is used to divide the structure of the software into smaller distinct components while defining the relationship between them. The division makes software development more efficient because developers can concentrate on solving smaller problems at a time, given that the subcomponents of the architecture are defined and documented delicately. Quality attributes define the different requirements for the software, which are considered when designing the software architecture. Quality attributes could be for example performance, security and modifiability. The final architecture should include an overview of the quality attributes used and how they are used to solve the problem. The software architecture is supposed to be defining the fundamental elements and structures of the software at a higher level; it shouldn't specify the individual components too precisely and only consider the requirements for public interfaces of the components. Software architects are generally responsible for the design of the software architecture. [20, pp. 1-8]

Documentation is a key part of the design of the software architecture. The architecture must be defined precisely enough to ensure that the developers can program the software without the possibility of misjudgements and guessing how the software should work. Without a well-planned and structured architecture, there is a significant risk in the development phase of the software, that the software becomes cluttered and complicated, while also making further development of the software more difficult. Documentation of the architecture isn't only useful to the software developers, but also other software architects, would they partake in the further design of the architecture. Well defined documentation of the architecture helps new architects to understand the design of the software and enables them to continue the work of the previous architects. [20, pp. 9-16].

Planning and designing the software architecture requires thinking and consideration from multiple perspectives. Documenting the architecture from only one point of view could have the consequence of missing some of the critical requirements which have been given to the software during the initial specification. Views are used to represent the architecture from one aspect to assess the requirements and structure of the architecture. One single view is not supposed to include the whole architecture design, that will be accomplished by using multiple different views. A single view considers only a few quality attributes and defines the architecture from that point of view. The views could be made for example from a physical, a logical, or a deployment point of view. [20, pp. 22-25].

3.1 General

Designing software architecture can be a demanding task, especially for large software, if started from scratch. There is a big chance that the fundamentals of the problem have already been solved before by other architects, which would make solving the same problem, again and again, unnecessary and a waste of resources. Instead of reinventing the wheel, existing designs of the already finished software could be reused to mitigate the creation of the new software architecture. The most used design principles of software architecture have been generalized into multipurpose solutions called architectural patterns. They specify the fundamentals of the software architecture by defining the core elements and relationships, which are needed for the software. Further specification from the fundamentals of the architectural pattern results finally in finished software architecture. [21]

Selecting the architectural pattern for the problem the architect is trying to solve must be done deliberately because the selected pattern will define the rest of the design process. The properties of the prospective software must be compatible with the pattern and it shouldn't interfere with the desired functionality of the software. The architectural design shouldn't be forced into a specific pattern for the sake of just having a pattern, instead, the architecture should be reinforced by a correctly chosen pattern. An incorrectly chosen pattern could make the software architecture worse than not using a pattern at all if the pattern doesn't comply with the requirements of the software. Sometimes it is better to not use any existing architectural pattern if the architecture requires properties that none of the patterns could offer. It is also possible that one pattern provides some required properties for the architecture, while another pattern provides the rest of them. In that case, combining the two or even multiple patterns should be taken into consideration, if they could be combined into a structurally coherent solution. [21]

There are lots of patterns designed and defined throughout the last decades of software development. According to Clements *et. al.* the first time the term architectural pattern was used was in 1996 by Buschmann *et. al.* in their book about software architecture [20, p. 34]. After that, the number of patterns has grown to a degree that presenting every possible architectural pattern would be an unreasonable task. Four architectural patterns, which could be the possible solutions for the problem at hand, have been selected to be addressed more closely. The selected patterns are layered, event-driven, microkernel and microservice, from which each of those is widely used in different types of software. The patterns are selected based on their popularity in different applications.

3.2 Layered

Layered architecture is the most used architectural pattern throughout the software industry [1, p. 19]. It resembles the structures of the companies having hierarchies among the staff, which has made it a well-known and adapted pattern overall. The layered architectural pattern consists of multiple distinct layers of components, which each have their role in the software. The layers are illustrated as horizontal stacks according to the application logic and the requests are usually sent from top to bottom, while the so-called notifications, which are inputs from the layers on the bottom end of the layers, are sent from bottom to up. The layered structure of the pattern makes the specification and operation of different layers transparent because the roles and responsibilities are defined for each layer separately. [21] This separates the layered architecture from monolithic architecture, which consists of just one large component without clear distinct components, even though the two could be thought of as the same kind of architecture [22].

Single layers could consist of multiple different components, which can communicate with each other as required. The requests coming from the upper layers are pointed to the interface of the layer, which acts as a surface to protect the components from direct access from other layers. It would also be possible that the requests could be sent directly from component to component between layers. This would make the software work more efficiently because there would be fewer abstraction levels needed in the layers. The downside is that the changes made to specific components could affect other layers, which could lead to dependency problems. For the sake of simplicity, using the interface is considered to be the usual practice for the rest of this thesis. [21]

The hierarchical structure of the layers means that the data will be moving to and through the layers according to the needs of the software. It makes the data flow clear and easy to understand if the data is passed to the neighbouring layer. In a layered architecture, the layers are generally considered closed layers. Closed in this context means that the layers above and below cannot communicate together without the layer in between them passing the data through. The required interfaces are defined in the architecture for each layer to allow sending the requests correctly to the next layer. While the requests pass through each layer one by one, the responses and notifications follow the same route back to the source of the request or the destination. [22] In case of multiple notifications received and sent simultaneously, either the first layer or the layers in between the destination, could be used to combine the input data into a single message to decrease the number of single messages passing through [21].

In some cases, the data may be required to be sent to layers further away from the current layer. Passing the same data through multiple layers would require that each layer, which is needed to pass the data through, would need to offer the methods for that purpose. The problem is that if the data is just passed through without a need for handling and modifying the data, the additional pass-through functionality for each layer would complicate the code and structure without having actual benefits. Passing large amounts of data through multiple layers could also reduce the performance of the application. For those reasons, some of the layers could be defined as open layers. They are layers, which the requests coming from the upper layers can pass through completely by going directly to the next closed layer in the vertical stack. This removes the need for the layers in between the sending and receiving layers to forward the request. [21]

Layers of isolation is a key concept in a layered architecture. It specifies that changes made to the components or the interface of the layer shouldn't affect other layers, apart from the interfaces of the layers right below. This is supposed to make the different layers independent of other layers and thus help the development of new features to be more straightforward. Having multiple open layers compromises the concept because changes to a single layer would require changes to every other layer that could send requests to the layer that is being altered if the changes affected the interface. This would result in architecture with complex dependencies between the layers and thus make updating the software troublesome [22]. So-called service layers are usually structured as open layers if they were used to log the application activity. Because they would most likely be called from any other layer for comprehensive log statistics, passing the requests through every other layer wouldn't make sense. When designing the service layers, it is particularly important to make the right design choices for the interface right away, because otherwise the modifications later could be difficult. [21]

Having distinct layers consisting of different components instead of a single monolithic codebase and user interfaces to limit the communication between the layers makes the development of single layers more versatile. That's because other layers are not dependent on the contents of the layers and what happens inside them. If the public endpoints of the interface are considered across the software when making changes to the structure, the other layers can be left intact. This means that the testing can be done for each layer separately using testing mocks, which try to simulate the real production data. It also allows developing and releasing new versions of the layers independently of the other layers, if the public endpoints are not altered. Possible hardware changes would be relatively effortless to implement to an existing application using layered architecture because only the layer directly connected to the hardware would be needed to change. [21]

Using the same layers across different applications could be useful if the same features are needed in different environments. Standardizing the public endpoints would make it possible to develop the layer for one application and then use the same layers in another implementation without any changes. This would reduce the total amount of time required for development when considering all the applications which could use the same implementation of the layer. [21] Using standardized layers has also been shown to reduce

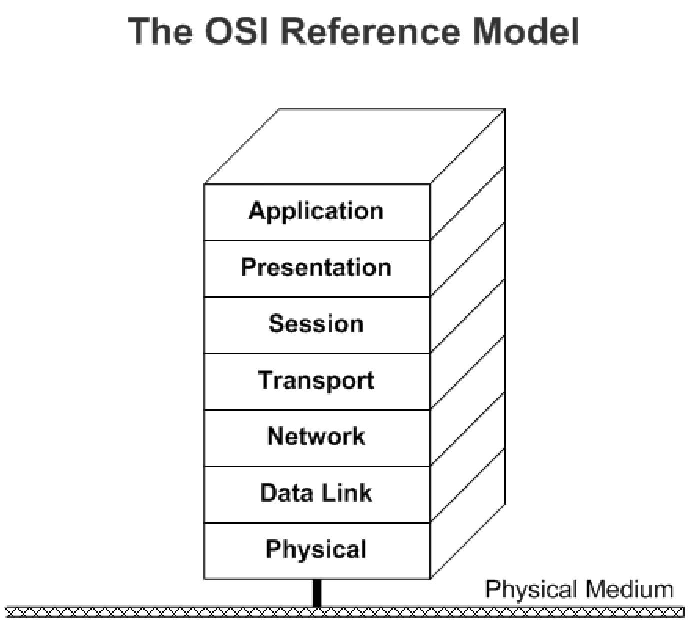

Figure 3.1. *Layered architecture in OSI model [24].*

possible errors made while programming the layer, according to empirical studies [23]. The standardized layers must be designed with caution because standardizing the layer to support a too wide range of features could affect the performance and maintainability of the layers negatively. If the different applications have too different needs for the layer, using standardized layers wouldn't be reasonable in those cases. [21]

Error handling in layered architecture can be done in multiple ways. The errors can be handled directly in the layers where the errors occur, or they can be passed through to a predefined layer, which handles all the errors occurring in the application. If the errors are handled locally in the layer they are occurring in, the data flow throughout the application stays simpler and more efficient. Especially in environments where the resources for the application are limited, a limited data flow could be required for the application to function properly. Passing through the errors from lower layers to higher requires additional handling for the error in the source layer of the error because in higher layers the detailed errors could be hard to comprehend and create additional confusion because of the low abstraction of the errors. [21]

The most prominent example of layered architecture in computing are the different networking protocols, which use different layers to specify the different parts of the communication functions. The different layers have been given descriptive names to describe the desired functions they are supposed to be doing. Most of the networking protocols follow at least loosely the OSI model portrayed in Figure 3.1. Some of the OSI layers might not be used, like in TCP/IP, but the remaining layers use the same fundamental concepts defined in the OSI model. Between the communication of two separate machines, the architecture consists of two identical network stacks, which are used to pass the message from one machine to another. The corresponding layers communicate with each other by using the lower layers to transfer the transmission. The standardized public endpoints of the lowest and highest layers ensure that different applications and physical transmission mediums can use the protocols effectively. [21]

3.3 Event-driven

An event-driven architectural pattern is mainly used, when the software is required to handle asynchronous communication in a scalable fashion [22, p. 11]. The communication is implemented by using requests and replies, which are sent throughout the software to fetch and retrieve the desired data [1, p. 20]. The pattern consists of multiple distributed and single-purpose components, which are used asynchronously to access the information needed in the application. The components can be added, relocated and modified dynamically, which adds flexibility to the pattern design. [22, pp. 11-12] Event-driven architecture can be implemented in a variety of different ways, from which mediator and broker topologies are the most used across the industry [1, p. 22].

Requests and replies in the event-driven architectural pattern are expressed as events. Events in general are things that change in each context, which are in the interest of the accountable. For example, an activated check-engine light in a car is an event, which tells that there is most probably something wrong in the engine of the car and has high relevance to the driver of the vehicle. In an event-driven architecture, the events are used to define the aspects of the occurrence, which include header and body parts. The header part contains the information about the details of the event, which are needed for it to be handled properly, such as a distinct event ID, the type of the event, name and timestamp. The body part is used to define what has happened in the event in question, which can be used to take actions by different components across the software that are interested in the specific event. Because the event-based communication method is used in the event-driven architecture, the communication is handled asynchronously. The events are triggered as they occur, so the communication can be very irregular depending on the application and the environment. [1, pp. 20-21]

Mediator topology, which is illustrated in Figure 3.2, is used in event-driven architecture to collect the events into a single event mediator, which sends the events to correct components. A single event could be wanted to be sent forward to multiple places if the event affects several components. It might be desired that the event would be sent forward in a correct sequence if the previous steps are needed to be handled before the event could be sent to the following places. Some events could be sent parallel to the destinations to be handled simultaneously if the events don't require processing from other components beforehand. The data flow in mediator topology starts from the event being sent to an event queue, which transports the event to an event mediator. The event mediator han-

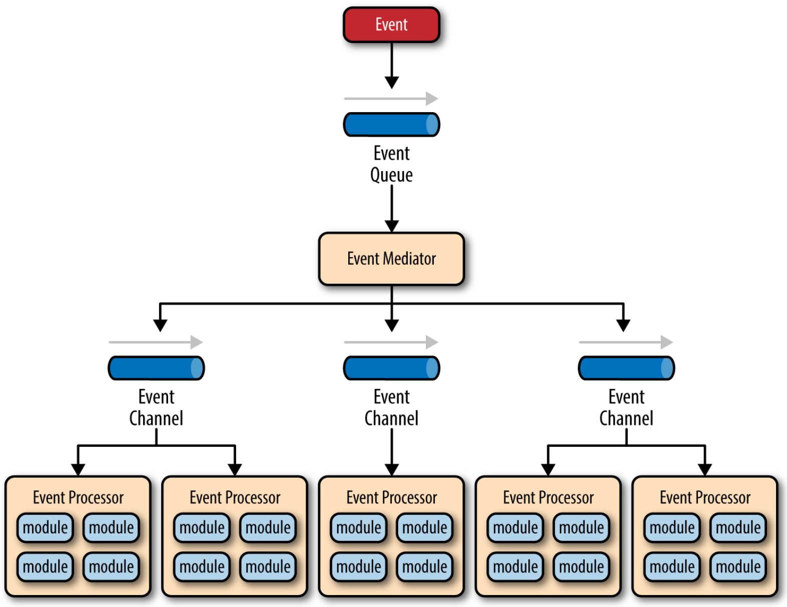

Figure 3.2. *Illustration of a mediator topology in event-driven architecture [22].*

dles the event and sends it to different event channels, which in turn use event processors to execute the desired functions for the event. [22, pp. 11-14]

A new event is initially sent from a client to an event queue, which is used to send the event to an event mediator. The number of event queues isn't restricted by the architecture, so it could be possible to have hundreds of different event queues to handle different kinds of events. The event mediator is used to handle events coming from event queues and the events created by it, and according to the needs of the event, it can determine the correct order for the events to be processed and sent forward to correct components. The events apart from the initial event are created in the event mediator after the correct order of tasks has been resolved from the initial event. The event mediator sends the events to event channels, which are used to execute the required functions regarding the events. [22, p. 12-13]

The event channels could be used to filter or pre-process the data received from the mediator if the same processing would be required to be done every time in the event processors anyway. This would decrease the amount of recurring code and make the application more straightforward. The actual logic for the event handling should be done in the event processors, so the event channels are not supposed to take part in that. [22, p. 11-12]

The event channels can be implemented as message queues or message topics, which affects the later parts of the communication stack. Message queues are used to send data from the source to the destination by placing the message into a queue, from which a consumer application then retrieves the message and removes it from the queue. Message topics are channels, where the clients publish the received data to be fetched by subscribing components. Multiple subscribers can fetch the same data from a single message topic, unlike in the case of message queues, where only a single consumer can fetch a single message. Because of that, the message topics are more often used in event-driven architecture to allow multiple event processors to handle the events parallel. Finally, the event channels transmit the events to event processors, which handle the actual computation of the data given in the event. A single event processor is supposed to handle a single task without the help of other processors to preserve the decoupled nature of the event-driven architecture. That is why a single event requiring multiple different actions could be handled by multiple event processors, either in serial or parallel, depending on the event. [22, p. 11-12]

Event broker topology suits cases, where the event flow remains moderate and relatively simple. The topology doesn't use any central event orchestration, like the event mediator, for handling and processing the events. Instead, they are sent through brokers to event processors, which handle the processing of the events. Because the broker is used to route the events to correct event processors, the client sending the initial event doesn't have to know about the specifics of the implementation beyond the broker component. The processed events are sent back to the broker from event processors to be sent either to the next event processor or back to the client as a response. [22, pp. 14-15]

The event broker consists of multiple event channels, which are used similarly as in the mediator topology. After receiving an event, the broker uses the event channels to send the event to a correct event processor and potentially altering the data of the event, if required. The broker itself doesn't have any control over the processing and scheduling of the events, it is used to pass through the data forward according to the header information of the events. The event processors handle the processing required by events and functions they are implemented to carry out like in the mediator topology. The difference is that after the tasks have been done, the event processor computes the next steps required for the current event. If further handling is required for the event by other event processors, the current event processor creates new events as needed to continue the operation. The newly created events are then sent to the event broker to be forwarded to correct event processors. After the event processor sends the new events to the broker and finishes the handling of the current event, it won't be involved in the processing of the current or any of the new events. [22, pp. 14-17]

The choice between the mediator and broker topologies depends on the complexity of the problem at hand. In general, a solution requiring a lot of parallel traffic and centralized event management for the correct order of execution, the mediator topology would be the right choice. The problem with the broker topology would be the lack of an event mediator, which greatly helps when multiple events are required to be scheduled concurrently in the correct order. On the other hand, a relatively simple problem without the need for highly parallel and scheduled tasks could be implemented using the broker topology. The mediator pattern could also be possible to be used, but it would add unnecessary processing through the event mediator if the events could be passed through event channels directly. [1, pp. 249-252]

The event-driven architectural pattern enables highly distributed systems with a good potential for scalability because the event processors are supposed to carry out single tasks and thus adding new event processors with corresponding modifications to event channels doesn't require a lot of modifications to the existing implementation. Asynchronous communication and the possibility to operate parallel make the pattern in principle perform effectively because the clients and event processors don't necessarily have to wait for a single event to be processed. Instead, new events could be processed at the same time by other event processors. Implementing tests for single event processors wouldn't require a lot of effort, because the event processors are supposed to perform single tasks for the events and thus making the use cases quite simple. [22, pp. 18-19]

Because of the asynchronous nature of the event-driven pattern, the design process for the architecture requires deep and undisputed knowledge of the environment for the solution to work properly. The sequences for the correct operation of the software could be difficult to sustain if one of the event processors would produce an error. That is because the following events after the failed one could require the desired functions to be executed, before proceeding forward, creating a chain effect of failures. Also, when the software consists of a large number of event channels for different tasks, the concurrency could become a problem, if some tasks couldn't be executed at the same time. This could be solved to an extent with the mediator topology, but having checks for every concurrent pair of events in a high dimensional application could make the mediator software very difficult to be maintained effectively. [22, pp. 18-19]

The event-driven architectural pattern is widely used in client-server communication use cases, where the client makes queries to various servers to get desired information or execute specific tasks. Instead of sending the requests to all the required servers one by one, the mediator or the broker patterns allow the tasks to be handled by a single request, which simplifies the client-side of the application. This is especially useful if multiple different clients are used for communication because all the operational logic can be implemented in the mediator or the servers, which act as the event processors. A single server is required to handle and route the communication between the client and the server, in the mediator pattern it handles the correct order of execution of the events, while in the broker pattern it just routes the events to the corresponding servers without further handling. [21]

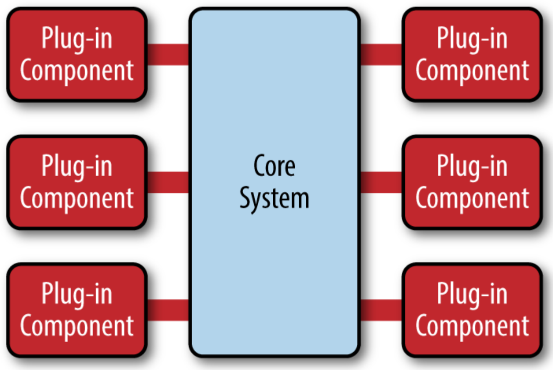

Figure 3.3. *Illustration of a microkernel pattern [22].*

3.4 Microkernel

Several types of software, like browsers or operating systems, require adaptability and the possibility to be updated after the initial release. New features could be requested to be developed to an existing system while retaining the existing functionality and design. Different customers could also want different features for the same product, which could make the updates burdensome if all the new features would be needed to be upgraded for every customer separately to keep the software up to date. The potential for future changes in the software should be taken into consideration when designing the architecture for software that is supposed to be upgraded later. Microkernel architectural pattern is one of the possible solutions for solving those problems. [21]

The microkernel architectural pattern is fundamentally an adaptable pattern, which can adapt to changes made to specifications after the initial specification and implementation. It was initially created for small operating systems, which could be updated later by adding new services to the existing solution. The upgrades could be done by adding new features like plug and play, which means that the existing software won't have to be modified for the new features to function. This can be achieved by using the same programming interfaces between the services to ensure compatibility with the existing solution. [21]

Microkernel pattern, which is portrayed in Figure 3.3, consists of a single core system and different plugin modules, which connect to the core via common interfaces. Clients can use the core to interact with the plugins to retrieve information and execute tasks the clients need for their purposes. The plugins can be either internal or external, depending on whether they are allowed to be accessed directly or not. The core is used to offer minimal functionality for the software to be operational while protecting the internal plugins of direct access from the clients. It is also possible for the plugins to communicate with each other if it is required by the application. The external plugins can directly communicate with the clients, for which a specific interface called adapter could be used to simplify the client-side of the system. [22, pp. 21-22]

The core system, which is also called a kernel, implements the core functionality of the software in microkernel architecture. It initializes the plugin modules and handles the communication principles of the whole software. The core manages the resource allocations across the plugins for the most optimal performance overall. The plugins can be connected to the core by using specific interfaces, which offer the communication endpoints for the plugins. The core isn't supposed to offer any complex functionalities, only the bare minimum for the software to be operational and the mechanics to add new features to the software as plugins. The abovesaid isn't realized in practice, because the performance of the software would suffer if everything functional was implemented to the plugins instead of the core. [21]

The plugin modules are components initialized by the core, which implement the specific functionalities for the software. For example, a software requiring connection to a hardware instrument could be implemented by using a plugin, which connects to the instruments and reads values from it. A client could request the readings from the instrument by sending a request to the core, which communicates with the plugin and sends the read values back. The plugins are independent of each other per se, although they may have some dependencies between each other for communication. The plugin modules can be split into two categories: internal and external plugins. [21]

The internal plugins are used to extend the functionalities of the core. They consist of features that won't possibly be needed for every build of the application, while also offering the division of the software into smaller functional pieces. The features that won't be needed for a specific use case are not initialized at all in that certain build of the application, which improves the performance compared to an application, which always contains every single feature built into it. Some features might be needed to be functional infrequently, for those situations the corresponding internal plugins could be activated run-time to keep the overall load of the application at the minimum. The internal plugins can be used to restrict the core from interacting directly with underlying hardware or software to keep the core simpler and more effective. If the internal plugins need to communicate with the clients, the core will be used in between to route the messages properly. This makes the internal plugins highly dependent on the core system. [21]

The external plugins offer additional services, which extend the features of the core system. They do not necessarily expand the core functionalities of the core system but offer other services and utilities to widen the scope of the application. Unlike the internal plugins, which are highly dependent on the core, the external plugins need the core just for the initialization and for communication preferences. The external plugins run on their separate processes, which makes them almost completely independent of the core. The core system offers the communication interfaces for the external plugins, where the clients can connect to. The actual communication between the external plugins and the clients occurs directly, the core is just used to create a new communication link. The responses from the external plugins bypass the core completely and are sent back to the clients. [21]

If the clients have previously used a service for their needs, that has been implemented to the new software using the microkernel architecture, it is possible that the public interface for the new software has changed and thus would require changes to the client-side. In the case of a complex service, the changes could be laborious to be made and it would be desired to find an alternative way to solve the problem. An adapter interface is used to translate the requests between the client and the public interface of the core. The client sends the request to the adapter, which then transforms the request to a correct format to be compatible with the core and possibly with the external plugins. The adapter can also be used to hide the implementation of the core system if the endpoints would change. [21]

The microkernel architecture excels in applications, where portability and the possibility to update the software during operation are required. The new features can be developed later as plugins to operate on the existing core and the upgrades can be made with minimal to no downtime. Updates to the existing plugins won't affect the operation of other parts of the application if the possible dependencies are not changed. The external plugins could be ported from already existing solutions to work on the microkernel application, and with the help of an adapter, the clients won't require any changes to be operational with the application. The independence of the plugins makes testing them individually a straightforward task because the features of a single plugin are supposed to be centralized into a specific problem and use case. The architecture allows the applications to be scaled horizontally as far as the resources of the machine the software is running on are sufficient, which makes the pattern very scalable. [21] The architecture also suits the cases, where smaller parts of the software could be implemented using the microkernel architecture and the overall design was implemented with another architecture. [22, p. 24]

The core system has to be designed and developed carefully because if the core must be maintained a lot after the initial deployment, the benefits of the microkernel architecture would be diminished. A large and complicated core could result in a monolithic type of application, with the addition of plugins that would add to the complexity even more. The architect must be careful with dependencies between the plugin modules when designing the software architecture because too many dependencies across the plugins could make the software cluttered and difficult to maintain afterwards. The microkernel architecture also provides worse performance, when compared to a monolithic application, because the additional layers of abstractions between the core and the plugins generate naturally more overhead. [21] Despite that, with good design choices the performance difference could be minimized, and the better maintainability could mean, that if a lot of new features were added to the software later on, the ease of development and the possibility to configure the software according to the needs of the end-user in microkernel architecture could result in better performance overall. [22, p. 26]

The microkernel architecture has been used to design the fundamentals of different operating systems (OS), like QNX and Integrity OS, which are used especially in the embedded systems. The kernel, which has the access to the processor and other hardware resources, consists of only the necessary software which is needed for the OS to function. The rest of the required functionalities for the OS are implemented as services, which are comparable to the plugin modules. The services communicate with the microkernel across different address spaces via inter-process communication (IPC) mechanisms through the different hardware components. This means that IPC is a crucial part of the microkernels in operating systems when it comes to the performance of the OS. [25] It is also the reason why most of the larger operating systems don't use the microkernel architecture and have a monolithic kernel instead [26].

3.5 Microservice

Some applications, like distributed systems, require high adaptability and scalability to operate and be maintained effectively. The microservice architecture tries to address those things by providing an architecture, which consists of multiple separate fine-grained services, which each handle a specific part of the application logic. The services are accessed via a gateway, which routes the requests from clients to the corresponding service, as in Figure 3.4. The services are completely isolated from one another per se, which makes the architecture pattern distributed and allows the services to be updated and deployed one at a time, while the rest of the application remains unaffected. [27, pp. 29-33] The microservice architecture is a relatively new concept when compared to other architectures. The term Micro-Web-Services was introduced first time in a conference by Peter Rodgers in 2005, which could be described as a predecessor to the term microservice [28]. Martin Fowler used the term "microservices" in a workshop of software architects in 2011, describing it as "suites of independently deployable services organized around business capability, automated deployment, intelligence in the endpoints and decentralized control of languages and data" [27, p. 30]. The microservices architecture is used in large scale web applications, like Netflix and Amazon, which use cloud-based microservices to provide the services they are offering [27, p. 52].

The microservice architecture can be designed using either asynchronous or synchronous communication methods. The synchronous communication method requires an active connection between the client and the service to handle the data transmission properly. This means that the instance in the client waiting for a response remains in a blocked state until the response is received or an error is triggered. Representational State Transfer (REST) is commonly used in applications requiring synchronous communication. It

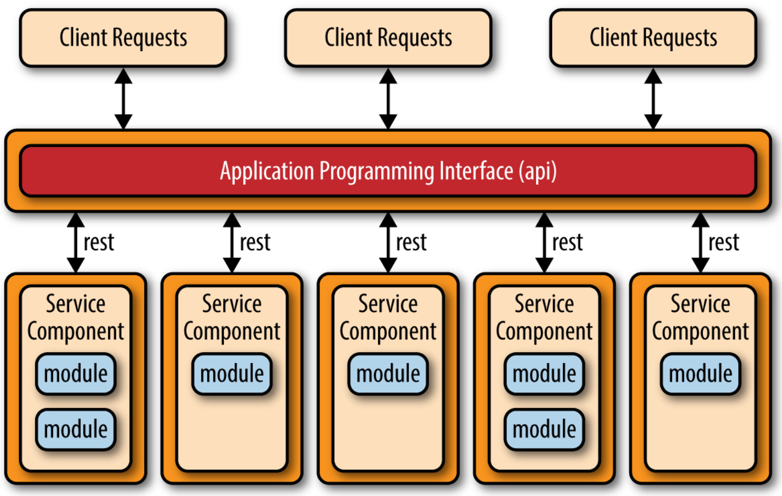

Figure 3.4. *Illustration of a microservice pattern [22].*

uses HTTP protocol to send the data between clients and services by considering the microservices as separate web servers and using URLs to target the correct services per request. [27, pp. 74-76]

In asynchronous communication methods, the client won't be waiting for the response from the service after the request has been sent, which prevents blocked states in the clients per se. Instead, it will be received by either of the following methods: a call-back function, a polling method or a message broker. The call-back function method uses a reference of a function implemented to the client by sending it to the service within the request. The server will invoke the function after the response is ready to be sent back and the client can then handle the request without being blocked during the waiting period. The polling method is more like a brute force approach when compared to the call-back method. The client will send the request to the service, which returns an acknowledgement after the request has been received. The client then starts to poll the service at a regular interval waiting for the actual response from the service while continuing the other possible tasks at the same time. The message queue method uses an additional message queue component between the client and the services to handle the data transmission. The client sends the requests to the message queue, which holds the requests until the corresponding service is available to handle the request. The client won't necessarily wait for the response, instead, the service could send the response back through the same message queue to the client to be processed on its own. [27, pp. 80-83]

The gateway is used to handle the communication between clients and services. The gateway accepts the incoming requests from the clients and sends them to the correct service components to be handled and processed. It exposes public endpoints for the clients to connect to the gateway. The gateway could be used to handle multiple subsequent calls to the services from one request from the client. This would be necessary if one service needed to access the information from another service or have it execute some tasks required by the initial service. All the required information about the whole execution plan of the services would then be returned as a single response to the client. The gateway is supposed to handle the security of microservice applications. Because the services are self-contained and the gateway connects to them using public endpoints, the authentication between every single service and the client would create overhead to the application. Instead of that, the gateway authenticates the requests beforehand and connects then directly to the services by using the security protocols defined in the services. [27, pp. 160-166]

Microservices are independent and separately deployed units that are used to handle a specific problem in an application [22, p. 27]. The independence of the services means that the services shouldn't have any dependencies between each other, which is made sure by decomposing the application into separate pieces. The services expose public endpoints for the gateway to connect to, which requires authentication to prevent unauthorized clients from connecting to the services [27, p. 14]. Limited functionalities of a single service might result in the need for multiple services to be used to handle a single request from the gateway. Combining multiple services for handling a single request is called service composition, which could be done either by service orchestration or service choreography. The service orchestration method uses a separate central coordinator component between the gateway and the services to determine the correct sequence of execution of the services. The service choreography works by distributing the responsibility of the service calls to the services themselves, which means that if a service needs another service to handle the task, it will call the other service directly. [27, p. 66]

The service components in the microservice architecture could be located in multiple different places, depending on the environment and the requirements for the application. The remote access protocol must know where to find the services to route the requests to the correct places, for which service discovery is used to solve the problem. The information about the services is saved to the service registry, which is a database used by the remote access protocol to connect to the services. The service registry could be implemented as self-registration or third-party registration methods. Self-registration is handled directly by the services when they save their metadata about the connection details to the service registry. The services will update the metadata at regular intervals to ensure the metadata will stay up to date. Third-party registration takes place in a separate component called service registrar. It keeps track of available services by fetching the metadata from them and sending the data to the service registry. [27, pp. 150-153].

Service discovery methods use the information in the service registry to find the correct servers for every use case. Two different service discovery methods could be used in a microservice architecture, which are client-side and server-side service discovery. The client-side discovery takes place in the client making the requests. The client must connect to the service registry to retrieve the metadata for the available services. In case multiple services are found from the service registry, the client performs load-balancing to find the service with the least load and sends the request to that service. The client must also fetch the metadata regularly in case the services get updated while the application is online. The server-side discovery occurs in the gateway, which handles the metadata retrieval from the server registry and keeps the information up to date. If load balancing is needed, it occurs also in the gateway, which means that the client doesn't have to know the specifics of the application, just the correct endpoint to the gateway for sending the request. [27, pp. 153-155].

The microservice architecture excels when it comes to the continuous development and deployment of applications. The division into smaller units having their own application logic means that the whole development process could be done separately for each of the microservices, given that they are not dependent on each other. [22, pp. 34-35] It also makes it easy to scale up the architecture, either by improving the existing microservices or adding completely new microservices with new features. The architecture is not dependent on running the whole application on a single hardware, instead, it could be run on multiple different hardware to increase the robustness and reliability of the application. The performance of the application could also benefit from running on multiple hardware by having the same services on different hardware and using a load balancer to optimize the flow of requests. The public endpoints of the microservices could be utilized by other microservices for their purposes, which could improve the reusability of the code. Because of the independence of the microservices, they could be developed with different programming languages within an application, which enables the use of the most suitable programming language for each microservice without compromising the others. [27, pp. 36-38]

The distributed composition of the microservice architecture affects the performance of the architecture because all the microservices would need to be run on their separate processes, which creates overhead to the application when compared to more compact solutions. The possibility to run the microservices on different hardware also requires the network between the gateway and the microservices to be operating at high speed, and even if that was the case, the performance would still be lower than by having all the microservices running on the same hardware. Even though the microservices could send requests to each other, the architect has to be careful by not overusing the concept, because it could create complex dependencies and complicate further development of the software. The overall complexity could also become a problem if the microservices would become too finely granulated because that would require heavy documentation to be maintainable. The possibility of using different technologies and the network between the microservices increases the chance for security vulnerabilities because even if the security would be initially at a good level, some of the used technologies could be exploited later. [27, pp. 45-46]

Microservices are widely adopted in use cases, where agility and ease of deployment are the key factors of the problem at hand. Different cloud applications are a prime example of those requirements because they are generally required to be updated frequently to stay competitive with the competition. According to Xia *et. al.* the microservices could be used in robotics to reduce the problems of limited storage space and computational power and allow advanced concepts, such as collective learning and shared knowledge-base to be implemented to the solution. The paper suggests that the subsystems of the cloud robot systems could be modularized into smaller microservices in contrast to a single monolithic type solution. [29] Ibarra-Junquera *et. al.* propose in their studies, that microservices could be a viable solution in industrial bioprocess applications. The different field devices provided their public endpoints and acted as microservices to be read and operated by human-machine interfaces and component controllers. The architecture was perceived as being flexible, scalable and robust and those claims were validated by using real hardware and process in testing. [30]

4. REQUIREMENTS AND SELECTION OF THE ARCHITECTURE

This chapter describes the practical environment and requirements for the device interface in the given context. The technologies used in the development of the new interface and the reasoning for using them are discussed, while also addressing the other possible technologies, which could have been used. The architectural patterns described in chapter 3 are evaluated considering the requirements of the environment and the most suitable of them is chosen as the base pattern for the new device interface. Finally, the possibility of combining some features from other patterns to the selected is considered and the final architecture for the device interface is determined.

4.1 Environment

In general, a device interface is needed for communication between a physical device and a system, which is using the device for its purposes. The requirements for such an interface would depend on so many different things regarding the environment and the needs of the whole system, that it would be practically an endless task to list and analyse them all. Instead, this thesis limits the scope of the environment into a specific use case, where the underlying system consists of a database, which other external systems use to handle and process the information stored in it. The devices handled by the device interface are limited to different devices found in the industrial environments of power plants, such as PLCs and sensors for example. The device interface is supposed to connect the devices to the database by reading and sending values between them using the suitable communication methods available.

The system, which connects to the interface, is a database in this context. The database is operating in a server machine, which is physically located in the same power plant as the devices are. The device interface is used to retrieve and send the required information between the database and the devices to allow the operation. The values in the database are then handled correspondingly to the actions taking place in the plant. The database doesn't send requests directly to the device interface, instead, the request is handled by marking the desired field as being sent, while the interface polls for changes in the corresponding table and takes action, when a marked row is detected. This means that effectively the device interface has to maintain a connection to both the database and the physical devices.

The field devices found in the environments consist of different I/O-devices, which support either reading, writing or both. The devices support different communication methods, which the device interface could use for communication. More modern devices generally use TCP as the communication method, which suits more complex applications. The devices are identified by the IP address and the port when using TCP communication. Simple devices could use serial communication to communicate with the device interface. The devices could in theory be connected directly to the local server by a hardware serial interface, but in reality, external serial interface devices using TCP are used to decrease the lengths of the serial cables and also to protect the server machines from power surges.

4.2 Technologies

The new implementation of the device interface will be done by using C# as the programming language with the help of the .NET Core computer software framework. C# is an object-oriented programming language, which supports strong type checking, detection of uninitialized variables and automatic garbage collection. It is supported by the .NET Core framework, which is a cross-platform software framework consisting of different tools and libraries for developing software applications. [31] It handles the memory management, exception handling and type-safety of the applications with the help of the Common Language Runtime (CLR) execution engine. .NET Framework Class library is used by .NET Core to provide the common functionalities, such as types and APIs, for developing the solutions. [32]

C# and .NET Core are not the only technologies that could be used in developing the device interface. C++ is a programming language, which supports object-oriented, procedural and many other programming paradigms. It can be compiled directly to a native code of the machine, which makes it a very efficient programming language. .NET Core framework could be utilised with C++ to develop software, but other frameworks, like Qt, could also be used. [33] LabVIEW, which was used for the target company's previous version of the device interface, is a graphical programming language developed by National Instruments. It is suitable for uses cases, where the software has to interact with hardware, so it could be used again for the new implementation. [34]

There are a lot of different technologies available in addition to the abovesaid technologies, but C# with the .NET Core framework were selected beforehand to be used for the new device interface. C# is powerful enough to handle communication between the database and the devices, while also being relatively straightforward and user-friendly to use, thanks to the automatic garbage-collection and ready-made libraries of the .NET

Core framework. The target company has started using those technologies in other projects during the last few years, which was the main reason for using them. Because the knowledge of the technologies is widespread across the company, it makes the development effortless and more reliable, when compared to using technologies with limited knowledge beforehand.

4.3 Requirements

Requirements for the device interface are presented in Table 4.1. Given the scope of this thesis, a device interface is software, which should be designed and developed to be as lightweight and robust as possible. The basic concept behind the device interface is to read and send commands to the field devices and return the responses to the database. In practice, the functionality of the interface should remain as straightforward as possible to avoid any possible performance issues, because of the simplicity of the functions it performs. If the device interface would perform well enough, it could operate in a low-cost embedded device, with the sole purpose of running the device interface. Thus, the performance (R1) of the device interface is the most important factor, when designing the architecture.

Maintainability (R2) signifies the ability of the software to be updated and repaired while operating in the production environment [35]. It is an important factor of the industrial device interface because low maintainability could result in long downtimes of the environment. Good maintainability ensures effective operation of the device interface by minimizing the downtimes and enables quick response times for fixing possible errors found in the software. Diverse environments also require different device drivers for the interface to be operational. A small power plant could fare with a few supported devices, while larger plants could require a dozen of device drivers. Configurability (R3) is one of the important factors of the device interface because it allows the interface to be efficient in each environment while supporting only the required field devices.

Safety and security are qualities, which are necessary for the device interface to operate reliably while protecting the data from possible third-party attacks. Safety (R4) is used to describe the capability of a system to tolerate faults, which are caused by the system itself [36]. The faults are caused by unintentional events, which could for example consist of different errors caused by defective parts of code or inadequate error handling of those defective parts when dealing with software. Security (R5) on the other hand describes the ability of the system to tolerate faults caused by the surrounding environment or external parties [36]. Those faults could consist of erroneous data retrieved from external systems, or an external party trying to cause faults or steal data from the software.

Table 4.1. *Requirements for the device interface.*

Index	Requirement
R1	Performance
R2	Maintainability
R3	Configurability
R4	Safety
R5	Security

4.4 Selection and analysis

The architecture for the device interface will be designed based on the requirements defined previously in this chapter. Chapter 3 introduces four different preselected architectural patterns, which will be evaluated and compared regarding the qualities they offer. The requirements specified in the previous section are used in the evaluation of the architectural patterns, performance is evaluated by dataflow and the physical location of the field devices. The most suitable architectural pattern based on the previous comparisons will be used as the basis for the overall architecture, while also considering the possibility of combining some concepts from other patterns to the final design.

4.4.1 Dataflow

Dataflow of the environment on the server's side consists of reading and writing information to the database and making the respective requests to the physical devices. The communication between those two endpoints should be kept as simple as possible to avoid any unnecessary overhead and latency to the data transmission because those could affect negatively the overall performance of the environment. The layered architecture uses distinct layers to split the functionality of the components into smaller parts. The communication between the topmost and the lowest layers requires the data to pass through all the layers in between them, bypassing the possible open layers. If the software would have lots of open layers, the performance of the data transmission could be quite good. On the other hand, if the data would pass through all the layers found in the application, it could generate quite a lot of overhead, when compared to the architectures with more straightforward dataflow.

The event-driven architectural pattern handles the data flow by transferring the data via events, which could be triggered either from outside through an API or internally from the application logic. The sequence of the data transmission in general from the database to the devices could be thought of as event-driven when considering the cases where a request is sent from the database either by the user or the system itself. That's why the final architecture of the device interface will implement at least some concepts of the event-driven architectural pattern. The performance of the data transmission depends on the chosen topology. The mediator topology uses an event mediator to process the data and handle the correct sequence of the events. In complex cases, where the data is required to go through multiple different event processors and possibly even parallel, the mediator topology would be a good choice. The problem is, that the event mediator adds complexity and some overhead to the software because it has to support the aforementioned cases. The data flow in the environments is quite simple because a single request is targeted at a single device. That's why the broker topology would suit better for this case because it removes the need for an event mediator completely. If the architecture would have support for decoupled components, the data transmission between the broker and the event processors would need to use public interfaces for connection, which would be slower when compared to direct calls within the application, like in layered architecture.

The data flow in microkernel architecture would occur through the single-core system, which connects to the plugin modules, which in this case would be the device drivers. Because the core system isn't supposed to have any complex logic and processing of the request, the data transmission inside the core would be good. Depending on the implementation, the performance of the data flow between the core system and the plugins could differ a lot. If internal plugins were used as the device drivers, the communication would be on par with the layered architecture, because the data could be transmitted using direct function calls within the application without a need for external services. Even though the performance wouldn't quite match a monolithic application, it would still be on top of the selected architectural patterns. In the case of the external plugins, the situation would be different, because the data transmission would need to occur through public endpoints. This would slow the data transmission similarly as in the event-driven architecture, which is not desired.

The microservice architecture handles the data transmission via a gateway to the independent microservices. Because the microservices are completely separate units from the gateway component, the data transmission requires the usage of public endpoints and thus additional communication methods, which creates the same kind of overhead as the previous architectures with the need for public endpoints. Depending on the communication methods used, the performance could be the slowest of these architectural patterns. The synchronous communication mode and the call-back method from asynchronous communication modes would perform similarly to the rest of the architectures requiring public endpoints. Using the asynchronous polling or message broker methods could slow the data transmission even further, because of additional data handling mechanisms creating more overhead.

4.4.2 Location of devices

The location of the field devices regarding the machine, where the device interface is operating, affects the evaluation of the architectural patterns greatly because different architectures perform better in different environments. For example, the devices could be connected to separate servers in separate networks, which would encourage to use distributed architectures. In the context of this thesis, the field devices are presumed to be located in the same local area network and physical location as the server machine. Software developed by using the layered architecture consists of multiple layers, which would all be located in the same server. It means that some overhead is applied to the software by the communication between the layers, and the amount of overhead is determined by the number of open layers found in the software. More open layers would reduce the overhead directly because the internal communication in the software would be lesser.

The event-driven architecture falls to the middle ground of the architecture patterns when it comes to the location of the devices. Because the devices are located in the same place as the server running the application, the distributed nature of the event-driven architecture doesn't offer any benefits from this standpoint. Both the mediator and the broker topologies would function similarly because the event processors would be located in the same server as the rest of the components of those topologies. As for the dataflow point of view, the support for decoupled components affects the performance of the architecture, when it comes to the location of the devices. Requiring public interfaces to connect to the decoupled event processors in the same server would create unnecessary overhead to the application without gaining any benefits.

The applicability of the microkernel architecture in cases, where the devices are located in the same place, depends highly on the plugin types used. Internal plugins are supposed to be implemented in the same location as the core system, which means that communication between the plugins and the core system would naturally perform well in this use case. The core system could use the internal plugins by direct function calls, which wouldn't affect the performance at all when compared to monolithic approaches. Using the external plugins would be a completely different case. The external plugins would be useful if they were located in other servers and there would be a need to connect to the plugins via a public interface. However, that is not the case, so requiring to use a public interface would slow down the performance of the application similar to the decoupled case of the event-driven architecture.

The microservices architecture is by definition a highly distributed architecture, which would be excellent for distributed environments. The problem is that the performance of the architecture is the worst of the four handled in this thesis when it comes to the applications operating on the same server. All of the communication modes require the gateway to connect to the microservices by using public endpoints, which is as slow as with rest of the architectural patterns requiring the public endpoints. Microservice architecture doesn't offer any other possibilities for communication between the gateway and the microservices, otherwise, it wouldn't be called a microservice architecture anymore. The lack of alternative solutions within the architecture for the communication, unlike for example in microkernel architecture, which supports the internal plugins, results in the worst applicability from the location point of view.

4.4.3 Maintainability and Configurability

Because the layered architecture consists of distinct layers, they could be updated separately without altering the rest of the application. The problem in maintaining software implemented using the layered architecture is the possibility of using open layers in between the topmost and lowest layers. Making changes to the layers next to the open layers would most likely require changes to the other layers as well. It would also slow down the deployment of the software because multiple layers would be needed to be updated. While the layered architecture scales well vertically, the problem is that it doesn't help, when developing and configuring new device drivers. The device drivers would need to be implemented into a single horizontal layer, which would make the driver layer very large and the public interface for the other layers would become clustered. The configurability in such a case would be minimal to non-existent. If the device driver layer would be modified separately for every use case, multiple different versions of the layer would be required to be created, which would make maintaining the different versions very troublesome.

The event-driven architectural pattern is a distributed pattern, which supports separate components within the software. The possibility to deploy the components individually makes it a good alternative when it comes to the maintainability of the software. Changes to event processors wouldn't affect the components of the software, which would only cause downtime for the specific feature the event processor is handling. Because the event processors handle the actual functionalities and communication with the field devices, they are usually the components, which require modifications. Nonetheless, if changes were made either to the event mediator in the mediator topology or the event channel in the broker topology, downtime would be required for the whole software, because they are the interfaces between the event processors and the database. The configurability of the pattern depends mainly on the implementation of the event mediator or the event channel. Because the event processors could be configured and developed separately, the key factor is that could they be added to the existing solution without changing the mediator or the channel. If the public interfaces were initially made configurability in mind, it could be possible.

The microkernel architecture performs similarly with the event-driven pattern when it comes to the maintainability of the software. Both the internal and the external plug-ins could be updated on runtime, which would make the downtime almost non-existent. The core system is the central component of the microkernel architecture, which handles the basic communication with the external system and the operation of the plugins. If it would need to be updated, the whole software would be down while the updates were taking place. The internal plugins are highly dependent on the core system, so it should be considered if the core would also be needed to be updated when making changes to the internal plugins. The external plugins on the other hand are just initialized by the core system, so updating them wouldn't most likely require any changes to the core. Configurability is inherently supported by the pattern, because of the plugin modules. The concerns expressed, when addressing the maintainability aspects of the pattern, also apply to the configurability. If the core system requires changes, it would be more laborious to develop new features, when compared to the cases, where only new plugin modules were added.

Regarding maintainability, the microservice architecture prospers from that point of view. Having the microservice components completely isolated from other components means that they could be updated without having any effects on the rest of the software. The only thing, which could indicate to the end-user of the software, that the microservices are being updated, would be if the requests were being sent exactly at the same time the update is taking place. The different communication methods don't affect the maintainability too much, even though the asynchronous methods would handle the real-time updates a little bit better, because of the nature of those methods. The gateway updates could in theory make the software completely unusable during the updates if for example the public endpoints would be required to be updated. However, because the gateway is used to just route the requests to the correct microservices, it could be duplicated to allow parallel operation, while the updates are being carried out. The configurability is also among the best when compared to other patterns. The microservices could be added on the fly to the environments and as long as the gateway supports the new microservice, the new device driver would be operational without further issues.

4.4.4 Safety and security

The safety of the layered architecture is not too good when considering the device interface. The problem in the layered architecture is, that the device drivers are located in a single layer. If one field device-specific driver would fail, it could cause the whole device layer to fail with it. Of course, this could be prevented by designing and programming the layer in a way that it wouldn't happen. This would however require additional effort to the design process and add complexity to the layer, especially when lots of different device drivers were implemented. When it comes to the security of the pattern, the layered architecture doesn't fare well in that case either. Because the layers are connected by public endpoints, it opens up the possibility for external attacks to be attempted at the software. All of the layers would need to be designed with that in mind, which could again increase the complexity of the application, while possibly reducing the performance.

The event-driven architectural pattern handles the safety of the software relatively well because the event processors are decoupled from the event mediator or the event broker components. The faults occurring in the event processors could be handled within the processors, which shouldn't affect the operation of the rest of the software. Faults in the event mediator or the event broker could result in the whole application failing if the faults were not handled correctly, so designing and implementing those components requires more resources to minimise the possible defects. Another problem could arise from chaining the events because if an event processor required by another component would fail, it could create a chain effect of multiple different components failing if the chain of events would be long. It means that components requiring other components must be able to handle the errors coming from elsewhere and pass the information forward. The security of the event-driven pattern depends on the communication methods used between the components. If public endpoints were used for communication between the components, the problems discussed in the layered architecture would also be present here.

The microkernel architectural pattern falls to the middle ground when considering the safety of the application. While the external plugins would offer naturally good fault tolerance towards the rest of the application, the internal plugins and the core system could be troublesome regarding safety. Errors in internal plugins shouldn't affect the rest of the software, but it could be possible because they are highly dependent on the core system. The core system must be designed in a way, that will catch all the possible faults coming both from the core and also the plugins because otherwise, the whole software could become non-operable. Good fault tolerance of the core system might come at the cost of performance, which could also affect the performance of the internal plugins. Microkernel pattern handles security well when compared to other patterns if the internal plugins were used. Because the internal plugins are used directly via the core system, it reduces the possible security risks a lot, when compared to using public interfaces for communication. The external plugins won't have the same advantage as the internal plugins, because of the usage of public interfaces.

When it comes to the safety of the software, the microservice architectural pattern is the best of the patterns addressed in this thesis. The microservices are completely isolated from other components, which means that errors occurring in one microservice won't affect the others. The microservice must handle and send the error response back to the gateway, which then sends it back to the underlying system. One microservice should represent one device driver, which would isolate the device drivers and thus increase the reliability of the error handling and recovery. Considering the security of the pattern, it is among the worst of all the patterns handled, because public interfaces are used for communication between the gateway and the microservices. Designing the microservices while considering security could reduce the overall performance of the application, like in the rest of the patterns using public interfaces.

4.4.5 Selection of the most suitable pattern

The layered architectural pattern suits well for cases, where vertical scaling of the software is desired. Unfortunately, the device interface doesn't benefit much from vertical scaling, instead, horizontal scaling would be desired for the device drivers. The layered architecture doesn't scale horizontally at all when considering the specifications, which reduces the applicability of the pattern for the device interface. The lack of horizontal scaling makes the maintainability and configurability bad because effectively the whole application would need to be shut down during the updating process. For the same reason, the error handling capabilities are not too good either, because every device driver is located in a single layer. The performance of the architecture regarding the dataflow isn't great if open layers were not used and there were lots of layers within the application. The device interface doesn't necessarily require lots of vertical layers, which would help with the performance losses coming from communication between the layers. Nonetheless, for the aforementioned reasons, the layered architecture doesn't seem to be the best possible choice for the device interface.

The event-driven architectural pattern is good for complex and large-scale problems, where the distributed nature of the pattern benefits the solution. While using different device drivers can be thought of as a distributed problem, the scale and the complexity of the problem are not that high. The communication between the underlying system and the devices consists of straightforward queries towards the devices, which means that completely event-based architecture would be a too complicated solution to the problem. The separated event processors result in relatively good safety and maintainability if the mediator or the broker won't be affected. The applicability for the rest of the qualities depends highly on the communication methods of the event processors, using public interfaces would potentially decrease the security and performance of the software, while on the other hand making development and maintaining the software less troublesome. The pattern could suit the device interface if a complex event flow was desired. However, in this case, it isn't the most suitable pattern, even though it handles the requirements better than the layered architecture.

The microservice architectural pattern excels in problems, where the software is desired to be split into multiple smaller parts running on their instances and also possibly on

Table 4.2. *Evaluation of the patterns by the requirements.*

	Performance	Maintainability	Configurability	Safety	Security
1.	Microkernel	Microservice	Microservice	Microservice	Microkernel
2.	Layered	Microkernel	Microkernel	Event-driven	Event-driven
3.	Event-driven	Event-driven	Event-driven	Microkernel	Layered
4.	Microservice	Layered	Layered	Layered	Microservice

different hardware. It is the best of the patterns handled in this thesis when it comes to the safety and maintainability of the software because isolation of the microservices makes them robust for error handling and single microservices could be maintained separately without affecting the others. Also, configurability is among the best, because only the necessary device drivers could be set up in different environments. Performance is the problem when it comes to using the microservices as the device drivers because they would need to be accessed by using public interfaces. It slows the software unnecessarily because the microservices would be located on the same machine. Also, the security of the pattern is among the worst of the patterns, because every microservice would need to be designed with that in mind. If the device drivers would need to be located in different locations, the microservice architectural pattern would be the choice for the device interface. Given the context of this thesis, it would add unnecessary overhead to the software without enough benefits to overcome the problems.

The microkernel architecture suits best at cases, where the application needs to be changed according to the needs of the environment while also providing sufficient performance. It offers the best overall performance of the patterns if the internal plugins were used. There would not be any restrictions, which could prohibit using them, because the whole software is running on a single machine. The security of the pattern is also the best with the internal plugins because they can be called from the core system directly. Maintainability and configurability are good, as long as the core system won't need to be altered in the process. The downside for the microkernel pattern is the fact, that the internal plugins are highly dependent on the core system, which could affect the safety and maintainability of the software if the core would produce an unhandled error. The external plugins have their upsides and downsides, but because using them doesn't offer many benefits over other distributed patterns, they won't be used. Overall, the microkernel architecture with internal plugins suits best for a device interface, which requires good configurability and performance, so it will be chosen as the main architectural pattern of the device interface. An overview of the patterns ranked from best to worst by the requirements is shown in Table 4.2.

4.4.6 Combining patterns

Architectural patterns are good general-purpose solutions, which work relatively well on their own for solving different problems. The problem in using a general predefined pattern is, that it might not be directly the best approach for a distinct problem, but instead, a more refined version of the pattern could be more suitable for a specific use case. Different patterns offer different distinct features, which are not strictly exclusive for one pattern but could also be applied to solutions using another pattern as a base. That is why the rest of the features from other patterns are evaluated and examined if they could be implemented to the microkernel architecture to enhance it to be a more applicable solution for the device interface.

The layered architecture offers the possibility to split the software into multiple distinct vertical layers, which implement a specific feature or functionality. While it could be applied to some extent to the core system of the microkernel architecture, it would add unnecessary complexity to the relatively simple part of the program. The core system is required to be able to connect to the underlying system and send data to it from the device drivers, which are connected to the core system as internal plugins. There are only a few required components within the core system that are needed, and also the dataflow won't necessarily benefit from the layered architecture, if the components were parallel rather than on top of the other. The layered architecture doesn't offer any features, which could enhance the microkernel architecture for this use case, so it won't be used to alter the design for the device interface.

The event-driven pattern uses events to transfer the requests from one component to another. The microkernel architecture doesn't necessarily take into account the format, which the requests and dataflow, in general, are using. The requests coming from the underlying system and new values received from the device drivers could be thought of as events, so it would make sense to use the event-based communication in the device interface. The reason why the event-driven architecture wasn't chosen as the main architectural pattern was that there hasn't been a need for a more complex event flow within the interface, so the benefits from using the events would be minor in the current situation. Nonetheless, using standardised events in a simple use case doesn't affect the performance of the interface too much, so it could be useful to have support for them if the more complex data flow is required in the future.

The microservice pattern offers the possibility to split the software into distinct components, which are used by the central gateway. When considering the microkernel architecture, the external plugins resemble a lot of the microservice pattern, because the core system acts as the gateway for the plugins. It has been considered before that using components with public interfaces would potentially slow the application and add possible security vulnerabilities without having clear benefits in this context. However, using the whole core system with the internal plugins as microservices, such that multiple core systems with specific internal plugins could be deployed to a single environment, would have the benefits of maintainability and good safety, while retaining the performance within the applications. The downside for that approach would be the overall requirements for the application because multiple core systems would most likely add to the performance requirements from the hardware. Overall, it would be an optional feature, which could be used in specific use cases and thus wouldn't add much complexity or affect the performance of a single-core system.

5. IMPLEMENTATION AND OBSERVATIONS

The final implementation and results for the device interface will be presented in this chapter. First, the overall architecture is opened up by describing the different components of the design, while comparing it to the selected architecture in the previous chapter. The implementation of the device interface is opened up by explaining the core features of the software in a lower level of abstraction. The implementation is then evaluated by its performance and resource usage by using the results of the practical tests done with the demo software. Finally, the possible drawbacks of the design are considered to recognize the use cases, where the design wouldn't be at its best.

5.1 Overall architecture

The overall architecture of the device interface is based on the microkernel architecture, which consists of the core system and internal plugins in this case. The core system is split into smaller components, which consist of different functionalities required by the environment. The internal plugins are used by the core system to connect to the devices by using the corresponding communication methods the devices support. The plugins are split into two components: the core system connects to a specific adapter component, which uses a more generic device driver component to connect to the device. The microservice architectural pattern is used partially to allow multiple core systems to be used parallel with each other by having unique identifiers for each distinct core system. An illustration of the architecture can be seen in Figure 5.1.

The core system of the device interface consists of three main components, which are considered as classes from now on. The classes are called ConnectionService, DriverService and StorageService. Each of the classes has its distinct functionalities and they are supposed to be run parallel so that the communication between the classes will stay as simple as possible. Additionally, external configuration files are used for different parameters required for the operation of the device interface. ConnectionService class handles the communication between the device interface and the database of the environment. It receives the commands from the database to be sent to the device drivers and sends the read values from the devices back to the database. The commands from the database are fetched by a specified interval, but also the interval for sending new

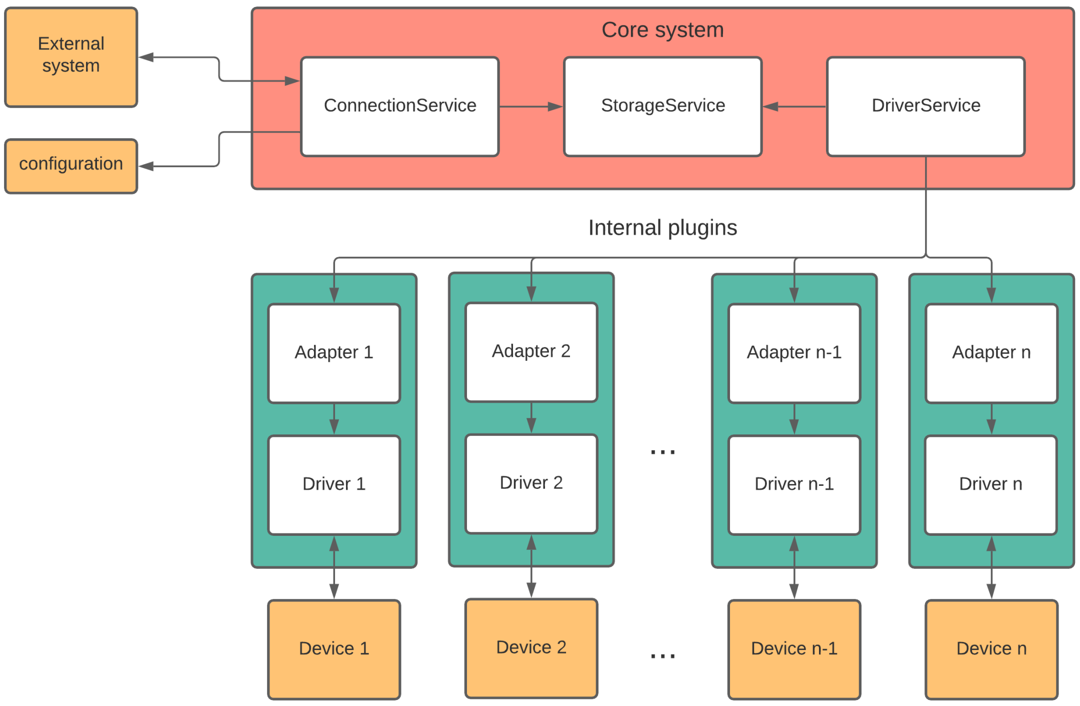

Figure 5.1. *The architecture of the device interface.*

values to the database can be configured so that continuous data transmission wouldn't cause any possible performance issues.

The DriverService class is used to connect and poll the device driver plugins. During the initialization of the device interface, the DriverService connects to each configured device driver and creates a new instance for each driver to allow independent operation of the device drivers and also to prevent any possible errors to be transmitted to other device drivers. Distinct instances handle the communication and connection to the devices by checking for the connection status of the device and try to reconnect to the device via the device drivers if the connection is lost. The DriverService uses the commands retrieved by the ConnectionService and sends them accordingly to the correct device drivers and returns the read values to be sent to the database.

ConnectionService and DriverService classes could in principle communicate directly with each other. This would make the device interface function fast and use fewer resources, but the problem is that it would create a high dependency between the classes, which could result in reduced reliability and also make the configurability of the device interface more difficult. To address those problems, StorageService class is used to store all the data that is used by the device interface. It acts as a storage for all the necessary configuration data for the device interface and also as a cache for the data received from the device drivers. The cache is required for the device interface to support specified intervals for sending the data to the database because the timestamps of the values can be used to determine when the new value can be sent.

The internal plugins are used as device drivers to connect to the devices and retrieve information from them. They offer the capability to control the devices via commands retrieved from the database and send the read values back. The internal plugins consist of two distinct parts, which are an adapter class and an actual device driver class. The adapter class is used to convert the data received from the device drivers to a uniform format, which the DriverService class can handle and send to the StorageService. Every internal plugin returns the data in the same format regardless of the type of the device, which results in a simpler and more maintainable DriverService class. The device driver classes contain the device-specific logic for connecting and communicating with the devices. The separation of the internal plugins into two parts was designed to make the device driver classes more general to be used in other applications.

The device interface isn't strictly limited to having only one core system per environment. Using the microservice architecture as a guideline, the environments could have multiple distinct core systems with internal plugins connecting to different devices. This would allow using multiple server machines to operate the device interfaces, which would increase the safety of the application. The core systems are separated by having unique identification for each core system, which are linked to the configuration parameters of the device drivers. One device could only be connected to one instance of a core system to prevent data errors caused by duplicated connections from different core systems.

5.2 Implementation

The implementation of the new device interface consists of the features specified in the overall architecture. The main components of the core system were implemented along with other classes and structures required to make the implementation more understandable and sustainable, such as universal error logging for every component of the device interface. Modbus master device driver was implemented to support communication with different Modbus slave devices, because of the high amount of such devices found in the environments. Modbus master devices have also the most complicated communication requirements when compared to other devices, which suit best for testing the safety and the performance of the whole implementation.

The main components of the core system are initialized during the start-up of the application in a so-called Worker-service class, which creates a service for each of the components. The services are operating parallel and only one instance is created for every component. StorageService is created as a singleton object, which means that it cannot be created more than once and it can be accessed by other objects by using public access points [37, p. 127]. In general, using singleton classes has been considered bad practice across the industry, because they create a global state to the application, which

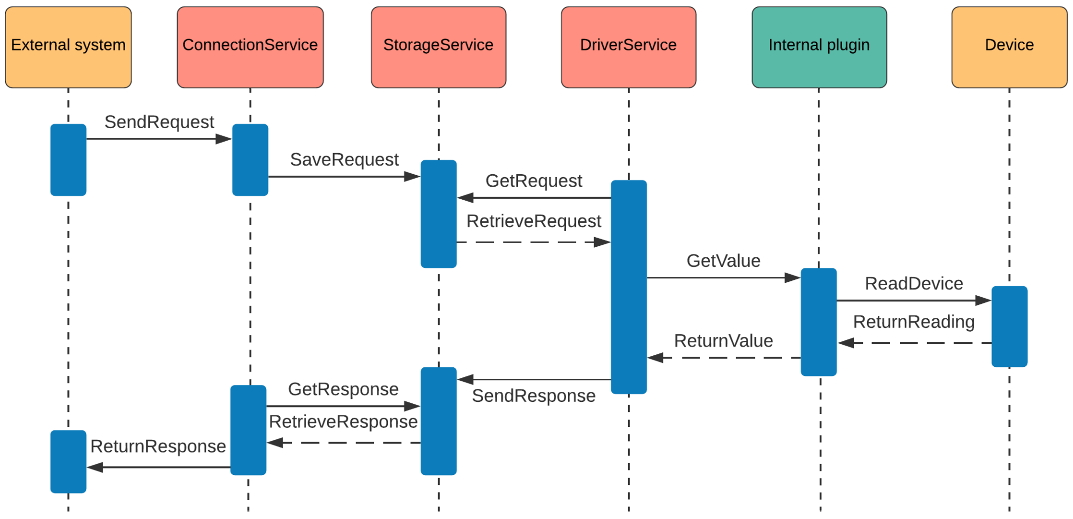

Figure 5.2. *The data flow of the device interface.*

could reduce the safety and testability of the software [38]. While that is partially true for StorageService, the reason for implementing it as a singleton class is, that it simplifies the data flow between the components. Safety is taken care of by implementing the public methods for the class in a way that a single method can only be called from one place. Also, the data stored in the StorageService is structured and accessed in such a way, that multiple simultaneous method calls from different device drivers won't alter the same parts of the storage.

ConnectionService class is also created only once per core system, but unlike Storage-Service, it cannot be accessed from other parts of the application. Instead, Connection-Service handles the connection to the database itself without the need for other components to interfere with the communication. The connection to the database is made by using TCP/IP protocol by connecting to the corresponding IP address and port. If the connection fails at some point in the operation, ConnectionService tries to reconnect to the database persistently. Initially, the configurations for the device drivers are retrieved from the database and saved to the StorageService object. After the device drivers have been initialized in DriverService, the commands are retrieved from the database and sent to the StorageService, while the newly read values from the device drivers are returned to the database. Every command and read value have a unique identifier configured in the database, which is used to match the information from the device drivers correctly.

DriverService class operates similarly to ConnectionService, only one instance is allowed, but other components cannot access it. DriverService handles the device drivers by initializing and communicating with them by polling them continuously. Required configurations for each device driver are retrieved from StorageService and sent to the device drivers while initializing them. The device drivers are initialized in their threads so that they can operate independently from each other. After the device drivers are initialized, DriverService starts polling them concurrently and if new data is retrieved from the device drivers, it is sent to the StorageService. Sending commands to the device drivers is implemented by polling StorageService for corresponding data retrieved from the database. If new commands are found, they are sent one by one to the device drivers, which will then either return the values requested from the devices or a status message containing information, if the commands were sent successfully or not.

The device drivers are split into an adapter and a driver. The adapter handles the conversion from application-specific configurations to a more general format, which the drivers use. The adapter doesn't offer any logic concerning communication with the physical devices but instead transmits the commands and requests forward to the drivers. Some function calls could be passed through without modifications, such as requests for asking for the connection status between the driver and the device. DriverService doesn't know the implementation of the device drivers, it only connects to the adapter, which handles the communication to the device drivers. The driver class handles the communication to the devices by using device-specific communication protocols and functions. Because the adapter class handles the processing of the configuration data, the driver can be completely generic without a need for supporting application-specific requirements. Most of the drivers have only a few functions, which the adapter calls for retrieving or sending information to the device. The data flow of the device interface for a single request from the database to the devices can be seen in Figure 5.2.

By using internal plugins as the device drivers, the architecture of the device interface would support dynamic usage of the device drivers, when they could be updated and added even during the operation of the application. DriverService class uses the same logic and functionality for every device driver regardless of the communication methods used by the device driver. This was achieved by using the adapter classes, which offer the same public endpoints for every single type of device for communicating with DriverService. Because the drivers can either receive or send information to the devices, the adapters offer general endpoints for both of those cases. This wouldn't necessarily allow real-time updates during the operation yet, instead of using a programming paradigm called reflection allows that. Reflection allows the manipulation of the application on runtime [39], which means that the device drivers could be altered dynamically during the operation. This would be possible with the standardized endpoints for the functions in the adapter classes. Reflection is used in the device interface by having a generic base-class for the adapters, from which all the actual adapters are inherited.

5.2.1 Modbus Master

The architecture of the device interface doesn't necessarily restrict the types of devices and communication methods for the device drivers. The implementation of the device drivers was limited to a Modbus master device, which suits testing purposes because of its complexity when compared to devices requiring only reading predefined data values as an example. The Modbus master device driver acts as a real Modbus master application, which could communicate with Modbus slave devices. While the Modbus master was the only device type implemented for the interface, the main concepts in the implementation apply to other device types also.

Adapters are used in the internal plugins to allow more generic implementation for the actual device drivers. The function calls for sending and receiving data are in a uniform format, which means that every adapter has identical public interfaces towards DriverService. Because the Modbus master driver is implemented by using the TCP/IP protocol, it requires only the IP address and port of the slave device for the initialization of the device driver. On the other hand, the requests for the slave devices differ a lot from other types of devices, which means that the function calls to the device drivers required customized functionality in the adapter. After the device driver has been initialized in the adapter, it can start sending the queries to the driver. Read and write requests are executed parallel to make sure, that slower operations won't block the faster.

Because the Modbus protocol uses a custom format for its header and payload while using the TCP protocol, the requests sent to the slave devices had to be parsed manually. The data for the requests was converted to binary representation so that the request could be sent directly to the slave devices. For the same reasons, the responses had to be parsed manually back to numeral representation. Possible errors received from the slaves could be detected from the response by comparing the function codes between the request and the response. If they were different, it would mean that the slave had sent an exception response and the exception code would be saved to the error log for further examination of the error.

Different Modbus function codes require different data to be sent in the request to the slave devices. Function codes 1 to 4 are different read requests, which all have the same format for the request [40]. Even though the data part in the response varies between the function codes, the implementation for the read request didn't require a lot of effort. Function codes 5, 6, 15 and 16 are different write requests [40], which were also implemented to the Modbus master driver. Implementation for the write requests required heavily customized functions for each type of write request because the format for the requests vary a lot. Some parts of the parsing could be unified between the function codes, such as calculating the number of addresses for a written request supporting multiple addresses and parsing the response data.

5.3 Evaluation

The new device interface supports parallel operation both between the device drivers and also within the drivers themselves based on the type of queries. This enables quick communication for the device interface across the drivers because they are not blocking each other. The speed of communication between the database and the devices was good across the different queries, which was made sure by the relatively simple data flow within the interface. The design allows intervals for different continuous queries to the drivers, which reduces the amount of data moving between the driver and the database and thus improves the overall performance. The architecture allows the intervals to be used only between the interface and the database so that the communication between the interface and the devices won't be affected. However, the intervals cannot be overused, because otherwise the dataflow could become too slow and cause problems in the application logic of the environments.

Error handling within different components of the interface ensured, that none of the thrown exceptions disturbed the operation of the other components. Connection problems are the most likely errors to occur in real environments, so they were tested thoroughly during the implementation. Losing the connection to the database didn't stop the device drivers from reading data from the devices to StorageService, and after reconnection, the operation resumed normally. The same applied to the connection problems with the devices, the operation wasn't disturbed after the reconnection. The required configuration files for the interface can be encrypted, so that critical data couldn't be stolen. Security is overall an important factor concerning the device interface, which is why it has been designed in a way that only the interface can initiate the communication between the database and the devices.

The resource usage of the new interface was generally low, which suits well for the environments. The usage of the processor in the test machine was low even with multiple device drivers running simultaneously. The memory usage stayed constant during the operation, which means that the implementation was done adequately and the garbage collector of .NETCore handles its job well. The device interface can be published as framework-dependent, which reduces the size of deployment packages and thus makes the deployment of the device interface quicker [41]. This comes at a cost of requiring the correct .NETCore version to be installed in the server, which could be problematic in some cases, but then a self-contained package could be used instead [41].

The maintainability of the device interface seems to be good concerning the future additions of new device drivers. The public endpoints for the base driver include all the necessary functionalities for new drivers to be developed on top of it without requiring changes to the core system. The architecture doesn't restrict the communication methods for the device drivers, which would allow using additional communication protocols besides the already implemented and tested TCP/IP, such as OPC UA. This could enable the usage of the device interface in a much wider range of applications because OPC UA allows the communication between different applications by standardized methods. One possible issue concerning maintainability is the use of adapter classes between the core system and the device drivers. While the adapters are generally used to transform the information between the two components, they are still making the codebase more complicated and thus increase the development time for new features or maintaining existing ones.

While the design and implementation for the core system were made as straightforward as possible, when considering the functional requirements for the device interface, new use cases could in theory require changes to the core. Every component outside the core system has been designed for the current version of the core, which means that making changes to the core could require a lot of changes to the adapters. This could produce additional problems because changes to the adapters could affect the operation of the device drivers, even though they wouldn't need to be modified directly. Another possible drawback of the design is the high overall dependency of the core system while using several device drivers through the same core. Even though the core system must be tested thoroughly and the possibility for errors is minimized, there is still a chance that something could break the core and thus affect all the device drivers associated with it. Using multiple core systems as a microservice architecture could reduce the problem, but it won't solve the possible issues completely.

6. CONCLUSION

The goal of this thesis was to research a suitable architecture for an industrial device interface. Common integration methods for the industrial environments were studied to find out possible requirements and qualities needed for the device interface. The architecture was decided by introducing a few preselected architectural patterns and comparing them to find the best alternative. The architectural patterns were studied from various literature sources and papers to gather up a dependable overview of each pattern. After the selection of the main architectural pattern, the possibility of combining features from the rest of the patterns to the main architecture was investigated, after which the overall architecture was decided. Finally, a demo software of the device interface was implemented and evaluated to find out how the architecture functions in practice.

The architectural patterns that were compared were layered, event-driven, microkernel and microservice patterns. While each of the patterns have their positives and negatives, the microkernel pattern was decided to be the best alternative from the group. The microkernel pattern allows the device interface to be configurable according to the needs of specific environments. This is achieved by splitting the architecture into two separate parts, which are the core system and plugin components. The core system provides only the necessary elements for the software to be operational, while the plugin components extend the functionalities of the core system. For the device interface, the core system was designed to handle communication with external systems and initialize the plugins, while the plugins were used to communicate with the field devices. The overall architecture was extended to allow multiple distinct core systems to operate parallel, which resembles the microservice architecture.

The overall architecture for the device interface was evaluated by implementing a demo interface for a single device type, which was a Modbus master device. The device type in question was selected because it has more complex communication functionalities when compared to other device types found in industrial environments. The implementation was tested against a Modbus slave simulator, which represented the functionalities of a real device. The performance and resource usage of the device interface was good, while it was also easily configurable to allow customization for different environments. Maintainability was at a good level for internal plugins, as long as the core system wouldn't need modifications. Overall maintainability could suffer from the high dependency of the core system because making changes to the core system could affect the plugin modules. Error handling of the architecture was good while testing, but if the core system would fail in any situation, it would also fail the internal plugins and make the whole interface inoperative. The possibility of using multiple core systems as microservices could help, but that would come at the cost of performance.

The architecture for the device interface was decided from four different architectural patterns. Lots of possibly suitable patterns for the device interface were left out of the thesis, which could be evaluated in further research. While the defined use case was somewhat specific, the overall architecture could still be suitable for any industrial environment, which requires multiple devices to be connected to an external system. The first two of the research questions were answered by using various literature sources, which allowed producing detailed answers for those questions. It was decided that the microkernel pattern was the most suitable architectural pattern for the given problem. The combining of the patterns turned out to be possible by incorporating qualities from the microservice pattern to the final design. The final question was proven by practical and empirical studies, which confirmed the conclusions of the first two questions. The implementation of the demo software for the device interface was successful, as the testing showed that it could fulfil the specified requirements. However, the scope for testing the architecture was limited to a simulator of a single device type, which meant that the practical applicability of the device interface couldn't be proven completely.

REFERENCES

[1] Raj, P., Raman, A. and Subramanian, H. *Architectural patterns: uncover essential patterns in the most indispensable realm of enterprise architecture*. eng. 1st ed. Birmingham: PACKT Publishing, 2017. ISBN: 9781787287495.

[2] Hookway, B. *Interface*. eng. Cambridge: The MIT Press, 2014. ISBN: 9780262525503.

[3] Science, I. C. *Device drivers*. URL: https://isaaccomputerscience.org/concepts/sys_os_device_drivers?examBoard=all&stage=all (visited on 09/11/2021).

[4] Hat, R. *What is an API?* Oct. 31, 2017. URL: https://www.redhat.com/en/topics/api/what-are-application-programming-interfaces (visited on 09/11/2021).

[5] Fall, K. R. *TCP/IP illustrated*. eng. Second edition. United States: Addison Wesley, 2012. ISBN: 0-13-280817-X.

[6] Braden, R. T. *Requirements for Internet Hosts - Communication Layers*. Request for Comments 1122. RFC Editor, 1989. 116 p. DOI: 10.17487/RFC1122. URL: https://rfc-editor.org/rfc/rfc1122.txt (visited on 10/19/2021).

[7] Zurawski, R. *Industrial Communication Technology Handbook, 2nd Edition*. eng. CRC Press, 2017. ISBN: 148220732X.

[8] *Industrial communication networks – Fieldbus specifications – Part 1: Overview and guidance for the IEC 61158 and IEC 61784 serie*. INTERNATIONAL ELECTROTECHNICAL COMMISSION. 2019. URL: https://cdn.standards.iteh.ai/samples/100312/214d4c8548084087b2c840d5e58591dd/IEC-61158-1-2019.pdf (visited on 10/19/2021).

[9] Organization, T. M. *MODBUS MESSAGING ON TCP/IP IMPLEMENTATION GUIDE*. Oct. 24, 2006. URL: https://www.modbus.org/docs/Modbus_Messaging_Implementation_Guide_V1_0b.pdf (visited on 08/31/2021).

[10] Mahnke, W. *OPC Unified Architecture*. eng. 1st ed. 2009. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009. ISBN: 1-283-94530-4.

[11] W3C. *Extensible Markup Language (XML)*. URL: https://www.w3.org/XML/ (visited on 10/19/2021).

[12] Hernandez, M. J. *Database design for mere mortals: a hands-on guide to relational database design*. eng. Addison-Wesley Professional, 2013. ISBN: 9780321884497.

[13] Dey, C. and Sen, S. K. *Industrial Automation Technologies*. eng. Milton: CRC Press, 2020. ISBN: 0367260425.

[14] Hanssen, D. H. *Programmable Logic Controllers: A Practical Approach to IEC 61131-3 Using CoDeSys*. eng. New York: John Wiley and Sons, Incorporated, 2015. ISBN: 9781118949245.

[15] Derby, S. J. *Design of Automatic Machinery*. eng. Vol. 182. Mechanical engineering. Baton Rouge: CRC Press, 2004. ISBN: 9780824753696.

[16] Cecilio, J. and Furtado, P. Architecture for Uniform (Re)Configuration and Processing Over Embedded Sensor and Actuator Networks. eng. *IEEE transactions on industrial informatics* 10.1 (2014), pp. 53–60. ISSN: 1551-3203.

[17] Balasubramanian, V., Aloqaily, M. and Reisslein, M. An SDN architecture for time sensitive industrial IoT. eng. *Computer networks (Amsterdam, Netherlands : 1999)* 186 (2021). ISSN: 1389-1286.

[18] Shin, S.-J. An OPC UA-Compliant Interface of Data Analytics Models for Interoperable Manufacturing Intelligence. eng. *IEEE transactions on industrial informatics* 17.5 (2021), pp. 3588–3598. ISSN: 1551-3203.

[19] Song, G., Zhou, Y., Zhang, W. and Song, A. A multi-interface gateway architecture for home automation networks. eng. *IEEE transactions on consumer electronics* 54.3 (2008), pp. 1110–1113. ISSN: 0098-3063.

[20] Clements, P. *Documenting software architectures : views and beyond*. eng. 2nd ed. SEI series in software engineering Documenting software architectures. Addison Wesley, 2010. ISBN: 1-282-76866-2.

[21] Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P. and Stal, M. *Pattern-Oriented Software Architecture - Volume 1: A System of Patterns*. Wiley Publishing, 1996. ISBN: 0471958697.

[22] Richards, M. *Software architecture patterns : understanding common architecture patterns and when to use them*. eng. First edition. Sebastopol, CA: O'Reilly Media, 2015. ISBN: 1-4919-7143-6.

[23] Zweben, S., Edwards, S., Weide, B. and Hollingsworth, J. The effects of layering and encapsulation on software development cost and quality. eng. *IEEE transactions on software engineering* 21.3 (1995), pp. 200–208. ISSN: 0098-5589.

[24] Microsoft. *Windows Network Architecture and the OSI Model*. Apr. 9, 2020. URL: `https://docs.microsoft.com/en-us/windows-hardware/drivers/network/windows-network-architecture-and-the-osi-model` (visited on 08/30/2021).

[25] Heiser, G., Elphinstone, K., Kuz, I., Klein, G. and Petters, S. M. Towards trustworthy computing systems: taking microkernels to the next level. eng. *Operating systems review* 41.4 (2007), pp. 3–11. ISSN: 0163-5980.

[26] Liedtke, J. Toward Real Microkernels. *Commun. ACM* 39.9 (1996). ISSN: 0001-0782.

[27] Surianarayanan, C., Ganapathy, G. and Raj, P. *Essentials of Microservices Architecture: Paradigms, Applications, and Techniques*. eng. 1st ed. Milton: CRC Press, 2020. ISBN: 0367249952.

[28] Habib, O. *A Quick Primer on Microservices*. Feb. 11, 2016. URL: `https://www.appdynamics.com/blog/engineering/a-quick-primer-on-microservices/` (visited on 06/18/2021).

[29] Xia, C., Zhang, Y., Wang, L., Coleman, S. and Liu, Y. Microservice-based cloud robotics system for intelligent space. *Robotics and Autonomous Systems* 110 (2018), pp. 139–150. ISSN: 0921-8890.

[30] Ibarra-Junquera, V., González, A., Paredes, C. M., Martínez-Castro, D. and Nuñez-Vizcaino, R. A. Component-Based Microservices for Flexible and Scalable Automation of Industrial Bioprocesses. *IEEE Access* 9 (2021), pp. 58192–58207.

[31] Metzgar, D. *.NET Core in action*. eng. 1st edition. Shelter Island, NY: Manning Publications, 2018. ISBN: 1-61729-427-6.

[32] Microsoft. *What is .NET Framework?* URL: `https://dotnet.microsoft.com/learn/dotnet/what-is-dotnet-framework` (visited on 07/04/2021).

[33] cplusplus.com. *A Brief Description*. URL: `https://www.cplusplus.com/info/description/` (visited on 07/04/2021).

[34] Ehsani, B. *Data Acquisition Using LabVIEW*. eng. Birmingham: Packt Publishing, Limited, 2016. ISBN: 1782172165.

[35] *Security and Resilience in Intelligent Data-Centric Systems and Communication Networks*. eng. Intelligent data-centric systems. San Diego: Elsevier Science & Technology, 2017. ISBN: 9780128113738.

[36] Bartnes, M. Safety vs. security?: 2006. ISBN: 0791802442.

[37] Gamma, E. *Design patterns : elements of reusable object-oriented software*. eng. 37th printing. Addison-Wesley professional computing series. Reading, Mass: Addison-Wesley, 1995. ISBN: 0-321-70069-4.

[38] Code, G. *Why Singletons Are Controversial*. URL: `https://code.google.com/archive/p/google-singleton-detector/wikis/WhySingletonsAreControversial.wiki` (visited on 08/13/2021).

[39] Malenfant, J., Jacques, M. and Demers, F. A Tutorial on Behavioral Reflection and its Implementation. (1996).

[40] Organization, T. M. *MODBUS APPLICATION PROTOCOL SPECIFICATION*. Dec. 28, 2006. URL: `https://modbus.org/docs/Modbus_Application_Protocol_V1_1b.pdf` (visited on 08/31/2021).

[41] Microsoft. *.NET application publishing overview*. May 2, 2021. URL: `https://docs.microsoft.com/en-us/dotnet/core/deploying/` (visited on 09/01/2021).