API-based Digital Twins

Architecture for Building Modular Digital Twins Following Microservices Architectural Style

Riku Ala-Laurinaho
API-based Digital Twins

Architecture for Building Modular Digital Twins Following Microservices Architectural Style

Riku Ala-Laurinaho

A doctoral dissertation completed for the degree of Doctor of Science (Technology) to be defended, with the permission of the Aalto University School of Engineering, at a public examination held at the lecture hall K1/215 of the school on 10th December 2021 at 12:00.

Aalto University
School of Engineering
Department of Mechanical Engineering
Mechatronics
Supervising professors
Associate Professor Kari Tammi
Department of Mechanical Engineering
Aalto University
Finland

Preliminary examiners
Adjunct Professor Mika Lohtander
Turku University of Applied Sciences
Finland

Assistant Professor David Hästbacka
Faculty of Information Technology and Communication Sciences
Tampere University
Finland

Opponents
Adjunct Professor Mika Lohtander
Turku University of Applied Sciences
Finland

Aalto University publication series
DOCTORAL DISSERTATIONS 159/2021

© 2021 Riku Ala-Laurinaho

ISBN 978-952-64-0593-3 (printed)
ISBN 978-952-64-0594-0 (pdf)
ISSN 1799-4934 (printed)
ISSN 1799-4942 (pdf)

Unigrafia Oy
Helsinki 2021

Finland
Abstract

The Fourth industrial revolution drives the rapid digitalization of industry. Digital twin is an emerging tool for digitalization and offers various features, such as simulations, predictive maintenance, and optimization of operation. These features are highly used case dependent, and at simplest, a digital twin may only be a metadata document of the physical product.

Because digital twins and their use cases vary greatly, creating an architecture that covers most of the digital twin features has been proven to be challenging. The digital twin features could be partly implemented using existing systems, but the current architectures do not employ these systems to the full extent. This dissertation proposes an Application Programming Interface (API)-based digital twin architecture for creating modular digital twins and presents a proof-of-concept implementation of the architecture for an overhead crane. The architecture enables using existing systems to implement digital twin and makes the digital twin data available from a single interface.

In the API-based digital twin architecture, digital twins consist of separate software blocks that implement the features of digital twins and can work in collaboration. Each block offers its services via API that is accessed via Data Link. Data Link acts as a message broker between the blocks and makes their services available on the Internet. The practical implementation of the broker is an API gateway, which forwards HTTP (Hypertext Transfer Protocol) messages to software blocks. This API-based architecture follows microservices architectural style, in which each software block is responsible for implementing one feature of a digital twin.

This study also compares Web APIs, REST (Representational State Transfer) and GraphQL, for industrial communication. These interfaces can be used to connect microservices to Data Link. In addition, the suitability of Web APIs for an additional interface to an OPC UA (Open Platform Communications Unified Architecture) server is investigated. OPC UA can provide access to an industrial machine PLC (Programmable Logic Controller) system. Finally, Open Sensor Manager (OSEMA) is presented as an enabler for data collection from a physical entity, which helps to keep a digital twin in sync with its counterpart. OSEMA allows remote software updates of sensors nodes and makes adding new ones easy by generating software based on the node configuration.

In conclusion, this thesis presented a general API-based architecture for digital twins. The proposed architecture makes digital twin data accessible, provides a platform for creating advanced applications, and allows the use of existing systems to build digital twins. In addition, the architecture is flexible, modular, and scalable. This thesis also compared Web interfaces, which allow connecting microservices to Data Link, in industrial communication and presented one microservice for a digital twin. This service is Open Sensor Manager that facilitates keeping digital twin sync with its counterpart. These contributions aid in the digitalization of industry and accelerate the Fourth industrial revolution.

Keywords digital twins, cyber-physical systems, architecture, REST, API, GraphQL, Industry 4.0
Tekijä
Riku Ala-Laurinaho

Välitöskirjan nimi
API-pohjaisten digitaalisten kokonaisuuksien korkeakoulutulostamisesta

Julkaisija
Insinööriratkaisuja

Yksikkö
Koneetekniikan laitos

Sarja
Aalto University publication series DOCTORAL DISSERTATIONS 159/2021

Tutkimusala
Mekatroniikka

Käsikirjoituksen pvm
18.06.2021

Välitöspäivä
10.12.2021

Viitettelyluvan myöntämispäivä
02.11.2021

Kielilähetys
Englanti

Monografia
Artikkeli

Essee

Tiivistelmä


Avainsanat
digitaalinen kaksosiksi, arkkitehtuuri, REST, sovellusrajapinta, Graphql, Teollisuus 4.0

ISBN (painettu) 978-952-64-0593-3
ISBN (pdf) 978-952-64-0594-0
ISSN (painettu) 1799-4934
ISSN (pdf) 1799-4942

Julkaisupaikka Helsinki

Painopaikka Helsinki

Vuosi 2021

Siivumäärä 150

My journey with digital twins started in 2018 when I joined professor Kari Tammi's group as a research assistant. I would like to warmly thank professors Kari Tammi and Petri Kuosmanen for seeing potential in me and providing me this opportunity. Prof. Kuosmanen also originally introduced me the concept of a digital twin and offered me the research assistant position. Since joining the group, Kari has guided me through the writing master of thesis and, now, he acted as instructor and supervisor for this doctoral thesis. I cannot thank him enough for all of the help and guidance he has provided me during the years I have been working with him. He has given me countless comments on the research I have been writing, been patient with my endless questions, and supported me through the difficulties. I have never needed to wait for a response when I asked for help for something. I would like to thank Juuso Autiosalo, the digital twin guy and the prophet of Digital Twin Web, who has an enormous knowledge of digital twins. When I needed an opinion for anything related to doing science, Juuso has been the first person I have asked for help.

I would like to express my gratitude to all of my co-authors Juuso, Joel Mattila, Anna Nikander, Jani Hietala, Heikki Laaki, and Kari, without their help this work would not have been possible. I would like to thank my pre-examiners, adj. prof. Mika Lohtander and prof. David Hästbacka, whose comments helped me to improve the quality of the thesis and make it more scientifically sound. I wish to thank Kaur Jaakma for sharing his knowledge about STEP standard. In addition, warmest thanks to "Terhojen istutus" group, i.e., to my fellow doctoral candidates Juuso and Janne Huotari, and our senior advisor Dr. Jari Vepsäläinen, who gave me invaluable comments and support during the writing of the dissertation. I wish to thank the great scientific writing teachers I had during my doctoral studies, Dr. Pia Lappalainen, Sonja Kniivilä, and Dr. Diane Pilkinton-Pihko.

I have been privileged to work with so many bright minds in our research group. I am grateful to all members of the Mechatronics group including people from ARotor Lab for creating a great atmosphere and bringing
joy to research work. I would like to especially thank my colleagues in Machinaide project Dr. Pauli Salminen, Joel, Juuso, Xinyi Tu, Chao Yang, and Tuomas Keski-Heikkilä. In addition, I would like to thank the other members of Machinaide project and the members of the previous DigiTwin project. I wish to thank Paula Thomsson-Levä and Merja Seppänen from HR for the friendly environment and for keeping my contracts in order. I would gratefully acknowledge Maria Chicherina who has taken care of the numerous expense reports from the equipment I have ordered. I would like to thank Tekniikan edistämissäätiö for the incentive grant.

Finally, I want to thank my friends for their support during my doctoral studies. Thank you Samuli, Ivar, Juha, Thomas, Joa, Konde, Atso, Laura, Timo, Emilia, An, Tommi, and Jessika. In addition, thanks to my family, Arja, Juha, and Vesa, for everything.

Espoo, November 15, 2021,

Riku Ala-Laurinaho
## Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Preface</td>
<td>1</td>
</tr>
<tr>
<td>Contents</td>
<td>3</td>
</tr>
<tr>
<td>List of Publications</td>
<td>5</td>
</tr>
<tr>
<td>Author’s Contribution</td>
<td>7</td>
</tr>
<tr>
<td>List of Figures</td>
<td>9</td>
</tr>
<tr>
<td>List of Tables</td>
<td>11</td>
</tr>
<tr>
<td>Abbreviations</td>
<td>13</td>
</tr>
<tr>
<td>1. Introduction</td>
<td>15</td>
</tr>
<tr>
<td>1.1 Objectives and scope</td>
<td>16</td>
</tr>
<tr>
<td>1.2 Research questions</td>
<td>17</td>
</tr>
<tr>
<td>1.3 Scientific contribution</td>
<td>17</td>
</tr>
<tr>
<td>1.4 Outline of the thesis</td>
<td>19</td>
</tr>
<tr>
<td>2. Literature review</td>
<td>21</td>
</tr>
<tr>
<td>2.1 Digital twin as a concept</td>
<td>21</td>
</tr>
<tr>
<td>2.2 Digital twin architectures</td>
<td>24</td>
</tr>
<tr>
<td>2.3 Research gap</td>
<td>28</td>
</tr>
<tr>
<td>3. Materials and methods</td>
<td>31</td>
</tr>
<tr>
<td>3.1 Microservices architecture</td>
<td>31</td>
</tr>
<tr>
<td>3.2 Digital Twin Document</td>
<td>33</td>
</tr>
<tr>
<td>3.3 Hardware</td>
<td>34</td>
</tr>
<tr>
<td>3.4 Technology stack</td>
<td>36</td>
</tr>
<tr>
<td>3.4.1 Protocols</td>
<td>36</td>
</tr>
<tr>
<td>3.4.2 Web Development</td>
<td>37</td>
</tr>
<tr>
<td>3.5 Algorithms</td>
<td>37</td>
</tr>
<tr>
<td>3.5.1 API gateway message forwarding</td>
<td>37</td>
</tr>
</tbody>
</table>
3.5.2 Generating software for a sensor node . . . . . . . . 38
3.5.3 Sensor node software update process . . . . . . . . . 40
3.5.4 Reading data from a sensor . . . . . . . . . . . . . . . 40
3.6 Measurement setup for analyzing network traffic . . . . . 42

4. Results and remarks 45
   4.1 API-based Digital Twin Architecture . . . . . . . . . . 45
      4.1.1 Architecture . . . . . . . . . . . . . . . . . . . . . . . . . . . 45
      4.1.2 Implementation for Ilmatar crane . . . . . . . . . . . 48
      4.1.3 Measurements . . . . . . . . . . . . . . . . . . . . . . . . . . . 49
   4.2 Web interfaces for industrial communication . . . . . . . 51
   4.3 Data collection . . . . . . . . . . . . . . . . . . . . . . . . . . . . 54

5. Discussion 59
   5.1 API-based Digital Twin Architecture . . . . . . . . . . 59
   5.2 Web interfaces for industrial communication . . . . . . 61
   5.3 Data collection . . . . . . . . . . . . . . . . . . . . . . . . . . . . 62
   5.4 Limitations . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 63
   5.5 Future work . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 64

6. Conclusion 67

Bibliography 69

Publications 79
List of Publications

This thesis consists of an overview and of the following publications which are referred to in the text by their Roman numerals.


Author’s Contribution

Publication I: “Data Link for the Creation of Digital Twins”

The author and Juuso Autiosalo designed the study. They were also responsible for developing concept of Data Link that connects digital twin features and makes them accessible. Digital Twin document was jointly developed by the author, Anna Nikander and Juuso Autiosalo. The user interface for Data Link was designed and implemented by Anna Nikander with the help of the author. The measurements were designed by the author and conducted by Joel Mattila. The manuscript was written by the author with the help of other authors. Kari Tammi and Juuso Autiosalo helped revising the paper.

Publication II: “Comparison of REST and GraphQL Interfaces for OPC UA”

The study was designed by the author and Juuso Autiosalo. The measurements were conducted by Joel Mattila and results were analyzed by the author, Joel Mattila and Juuso Autiosalo. The author compared REST and GraphQL interfaces and analyzed their suitability for OPC UA server. The used GraphQL interface was developed by Jani Hietala with the help of Heikki Laaki. The manuscript was written by the author and reviewed by Juuso Autiosalo and Kari Tammi.

Publication III: “Open Sensor Manager for IIoT”

Open Sensor Manager that allows managing sensors remotely over the Internet was designed by the author and Juuso Autiosalo. The author implemented the manager and conducted the user tests. The author wrote
the manuscript and it was reviewed by Juuso Autiosalo and Kari Tammi.
<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>Relations between publications and their contributions.</td>
</tr>
<tr>
<td>2.1</td>
<td>Hribernik et al. [23] proposed a product-centric approach (A) for information exchange between actors. In the traditional approach, each actor communicated directly with each other (B).</td>
</tr>
<tr>
<td>3.1</td>
<td>A monolithic system implements all functionalities in a single executable (A). In a microservices architecture, the functionalities of the system are implemented by separate services (B). Monolithic systems are scaled by replicating the whole system (C), whereas microservices architecture allows replicating only the necessary services (D).</td>
</tr>
<tr>
<td>3.2</td>
<td>DT document is a development effort to describe the metadata of a digital twin in a standardized way. This is a screenshot from the GitHub page of the standard draft (<a href="https://github.com/AaltoIIC/dt-document">https://github.com/AaltoIIC/dt-document</a>) showing an example of the document.</td>
</tr>
<tr>
<td>3.3</td>
<td>Ilmatar overhead crane that is located at Aalto Industrial Internet Campus.</td>
</tr>
<tr>
<td>3.4</td>
<td>OSEMA used Pycom microcontrollers along with I2C bus sensors.</td>
</tr>
<tr>
<td>3.5</td>
<td>Technologies used in this dissertation. White color indicates that the system was implemented as a part of this thesis, and red that it was not.</td>
</tr>
<tr>
<td>3.6</td>
<td>Overview of the OSEMA algorithms and the architecture of the system (Publication III).</td>
</tr>
<tr>
<td>3.7</td>
<td>The schematic of the measurement setup. The traffic between systems is mirrored to one of the router ports and Wireshark is used to capture the traffic.</td>
</tr>
</tbody>
</table>
List of Figures

4.1 Structure of the API-based Digital Twin Architecture (Publication I). ADTA provides a single interface to access digital twin features. ................................................................. 46
4.2 Screen capture of the Data Link user interface (Publication I). The user interface allows examination of digital twin metadata and features. ................................. 47
4.3 The feature that implements the needed functionality can be searched from the user interface (Publication I). ....... 48
4.4 Ilmatar digital twin consisted of several separate systems (Publication I). ................................................................. 49
4.5 Comparison of request execution times with and without the API gateway. The gateway adds a considerable amount of latency for the request execution times (Publication I). ................................................. 50
4.6 The payload sizes with one value (Publication II). min. = minified, w/o = without. ......................................................... 53
4.7 The payload sizes with 50 values (Publication II). min. = minified, w/o = without. ......................................................... 54
4.8 The user interface of OSEMA allows managing sensors (Publication III). ................................................................. 56
4.9 The usage roughness of the crane was examined with a three-axis accelerometer attached to the hook (Publication I). Students also investigated the sway of the hook with a similar measurement setup as a part of a mechatronics course. ................................................................. 57
4.10 OSEMA’s ease of use assessed by the students (Publication III). OSEMA was found reasonably easy to use. ............ 58
## List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Summary of the state-of-the-art review of digital twin architectures.</td>
<td>28</td>
</tr>
<tr>
<td>3.1</td>
<td>Configurable settings of sensor nodes (Publication III).</td>
<td>41</td>
</tr>
<tr>
<td>4.1</td>
<td>Request execution times in milliseconds for different requests and responses (Publication I). GW = API gateway, SD = standard deviation.</td>
<td>50</td>
</tr>
<tr>
<td>4.2</td>
<td>Comparison between the REST and GraphQL interfaces (Publication II). A column for OPC UA was added as a reference for industrial protocol. The values are relative to each other, not absolute values.</td>
<td>51</td>
</tr>
<tr>
<td>4.3</td>
<td>The request execution times in milliseconds for reading and writing values to the OPC UA server. SD = standard deviation. Disconnected and connected indicate if the OPC UA session was established before the request was made.</td>
<td>55</td>
</tr>
</tbody>
</table>
Abbreviations

ADTA  API-based Digital Twin Architecture
API   Application Programming Interface
DT document Digital Twin document
DTDL  Digital Twin Definition Language
DTW   Digital Twin Web
HTTP  Hypertext Transfer Protocol
IT    Information Technology
I2C   Inter-Integrated Circuit
JSON  JavaScript Object Notation
JWT   JSON Web Token
MQTT  Message Queuing Telemetry Transport
OPC UA Open Platform Communications Unified Architecture
OSEMA Open Sensor Manager
PLC   Programmable Logic Controller
PLM   Product Lifecycle Management
REST  Representational State Transfer
WoT TD Web of Things Thing Description
The era of the Fourth industrial revolution is here [1]! The Fourth industrial revolution, or so-called Industry 4.0, is invoked by digitalization and networking [2], decentralized production [1], and a shift from product-centric towards service-oriented industry [2]. Enablers for Industry 4.0 are low-cost sensors and data collection [3], smaller computers [2] and increasing computing power, availability of low-power networks, and Industrial Internet of Things (IIoT) including related technologies such as OPC UA (Open Platform Communications Unified Architecture). The change towards Industry 4.0 is driven by requirements for lower time-to-market, mass customization, flexibility, and efficiency of production [2]. The Fourth industrial revolution is part of the larger ongoing digitalization trend, the most obvious example of which are smartphones offering connectivity, sensing capability, and computational power. However, consumer products, such as smartphones, are well ahead of the industry in adopting Internet-enabled applications.

Digital twin is a tool for digitalization in the industry accelerating the revolution. It makes product data available and accessible which allows the creation of advanced applications, such as predictive maintenance, optimization of the manufacturing process, and analysis of the data for various purposes. The definitions for digital twins found in the literature are various [4]. Current definitions allow almost any “virtual” entity linked to an asset to be called a digital twin, such as the definition by Industrial Internet consortium [5]: “a formal digital representation of some asset, process or system that captures attributes and behaviors of that entity suitable for communication, storage, interpretation or processing within a certain context” and Autiosalo et al. [6]: “a virtual entity that is linked to a real-world entity”. Therefore, it is always necessary to define what is meant with digital twin in each use case. This thesis adopts the aforementioned inclusive definitions for digital twin by Industrial Internet Consortium and Autiosalo et al. In addition, the main task of a digital twin is considered to be providing access to product data via a single interface as proposed by Laaki et al. [7].
In addition to the absence of a unanimous definition for a digital twin, there is no widely accepted architecture for digital twins in the scientific community [8]. To provide a set of more general design guidelines for creating digital twins, Autiosalo et al. [6] proposed a conceptual-level Feature-based Digital Twin Framework (FDTF). The fundamental idea of the FDTF is to divide a digital twin into features, implement these features with separate software blocks that provide services via APIs (Application Programming Interfaces) following microservices architecture, and connect these features with Data Link to create a single digital twin entity. This thesis builds upon the FDTF and presents a general API-based architecture for digital twins and its prototype implementation. The API-based architecture allows the creation of modular, scalable, and easy-to-build digital twins suitable for various use cases. In addition, one of the main benefits of the architecture is that existing systems can be used to implement the features of a digital twin. This thesis also compares REST (Representational State Transfer) and GraphQL APIs in industrial communication, both of which can be used with the proposed API-based architecture, and introduces Open Sensor Manager (OSEMA) that allows managing sensors attached to the real-world entity remotely over the Internet. Sensors allow to keep the digital twin up-to-date with its counterpart. OSEMA offers a REST API and can be used as one of the features that construct the digital twin in the API-based architecture.

1.1 Objectives and scope

The objective of this thesis is to develop a digital twin architecture based on the conceptual Feature-based Digital Twin framework by Autiosalo et al. [6]. The research problem of this thesis is as follows:

*How to build an industrial digital twin that is modular, makes digital twin data available, and uses existing systems?*

Modularity is defined as the possibility of constructing a digital twin and its features from separate blocks based on the need. In addition to modularity, the other goals of the developed architecture are enabling the use of existing systems to build digital twins and making digital twin data and features available via a single interface. By following the architecture, the thesis aims to implement a digital twin for Ilmatar overhead crane [9] located at Aalto Industrial Internet Campus [10].

This thesis focuses on information-management-oriented digital twins in the industrial domain. The scope of this work – industry – is purposefully broad since the Data Link concept and the developed architecture are suitable for various digital twins in multiple domains. Furthermore, this thesis works in the intersection of mechanical engineering and com-
puter, communication, and information sciences, all of which are needed to implement a digital twin.

The scope was limited to digital twins of physical objects, leaving digital twins of abstract entities, such as organizations, out of the scope. Comprehensive analysis of the security of digital twins is also out of the scope of this thesis, but security was considered in the implementation of OSEMA.

1.2 Research questions

Based on the research problem and objectives, this thesis examines the following research questions (roman numbers indicate the publication in which the research question is examined):

1. How to build a digital twin in practice exploiting existing systems? (I)
2. How to make digital twin data and features available? (I)
3. What are the differences between REST and GraphQL Web APIs in industrial communication? (II)
4. Which one is more suitable to create API for the OPC UA server, REST or GraphQL? (II)
5. How to collect data from an industrial machine for its digital twin? (III)
6. How to manage data collection and change sensors' settings over the Internet? (III)

1.3 Scientific contribution

The scientific contributions of this thesis are revolved around the digital twin, divided into three more specific subtopics: introducing an API-based digital twin architecture, comparing Web interfaces, REST and GraphQL, in industrial communication, and enabling data collection from physical counterpart. These contributions are presented in Publications I, II, III, respectively, and discussed in more detail below. The relations between the publications and their contributions are shown in Figure 1.1.

In the scientific literature, no generally accepted architecture for digital twins exists [8]. Publication I investigated microservices-based digital twin architectures and presented an API-based digital twin architecture following the concept of FDTF by Autiosalo et al. [6]. The developed architecture allows the implementation of modular digital twins with a
low development effort. As a case study, the paper created a digital twin of an industrial machine with Ilmatar the overhead crane as a development platform. Finally, Publication I introduced a digital twin document, which contains metadata of a digital twin and information about its features. The digital twin document forms a basis of Digital Twin Web, a network of digital twins.

GraphQL is an emerging alternative for the dominating REST paradigm in Web APIs. It offers several attractive features, such as fetching multiple resources with a single request, control to which data is returned, and GraphiQL tool that offers Web user interface helping to inspect an API structure and write queries. The previous studies assessing the possibilities of GraphQL in the industrial communication are scarce. OPC UA is currently a de facto standard in industrial communication, but it lacks ease of use and hinders interoperability compared to Web APIs. Publication II compared REST and GraphQL APIs in the industrial domain and their suitability as an additional interface for OPC UA servers. In addition to analyzing the characteristics of the APIs, request execution times were measured, and the performance was compared. In the measurements, direct communication with OPC UA binary over TCP was used as a reference.

Reflecting and mirroring the state of the physical counterpart has been considered as an important feature of the digital twin [4], and it is claimed that the synchronization with digital and real-world should happen in real-time\textsuperscript{1} [11]. This dissertation does not aim to develop a real-time capable system but considers the freshness and timeliness of the digital

\textsuperscript{1}Real-time means that the operation is always completed within a specified time limit. However, there are several misconceptions about real-time [12] and real-time is often misused to describe any operation that happens swiftly.
twin data crucial. To have up-to-date data and reflect the real-world, measurement data needs to be continuously collected from the physical product. The contributions of Publication III are related to keeping digital twin in sync with its counterpart. It reduces the work to add new sensors and helps in the management of the sensors. The publication introduces Open Sensor Manager that enables software updates over the Internet for sensor nodes. These software updates allow changing the measurement settings and network configuration of the sensor nodes. Software updates can be initiated both via the Web user interface and REST API. The REST API allows connecting the manager to the digital twin using Data Link. OSEMA minimizes manual programming reducing the set-up time of new sensor nodes. The solution is based on mapping possible configurations and the related value-address pairs used to configure sensors via I2C (Inter-Integrated Circuit) bus to the database of the manager.

1.4 Outline of the thesis

This thesis is structured as follows: Chapter 2 describes the origins of the digital twin concept and examines the current digital twin architectures. The third chapter presents methods for building API-based digital twins, such as microservices architectural style, and the Web programming tools and frameworks used in this thesis. The results of this thesis, that is, API-based digital twin architecture, comparison of Web interfaces for industrial communication, and OSEMA are presented in Chapter 5. Chapter 6 discusses the findings and limitations of the work, and, finally, Chapter 7 concludes the dissertation.
2. Literature review

This chapter describes how the concept of a digital twin has evolved from twinning in NASA’s space program, software agents, and exact simulation of the product to the current form that allows various virtual entities to be called digital twins. For a more comprehensive review of the digital twin concepts and definitions, see for example [4, 8, 13, 14]. This thesis focuses on creating a modular digital twin architecture that makes digital twin data available and enables using existing systems, and the second section of this chapter reviews the state-of-the-art of digital twin architectures.

2.1 Digital twin as a concept

The origins of the digital twin concept can be traced back to NASA’s Apollo space program [15] between the 1960s and 1970s [16]. In the Apollo program, the space vehicles had a copy or copies of themselves on earth through the mission mirroring their condition [17]. The purpose of this – then physical – copy, or twin, was to assist in decision making during the flight by testing the consequences of actions beforehand [15]. This ability proved to be useful in critical situations, for example, when Apollo 13 crew was needed to be brought back home after the catastrophic explosion of the oxygen tank in 1970 [18]. In addition to assisting during-flight simulations, the twin was used in the training of the astronauts [17]. The physical twin used in the Apollo program had already several common attributes with a digital twin, such as: 1) remote twinning of a physical asset, 2) connection between physical and digital twin; there was a two-way connection between the twin and the vehicle, 3) adaptation to the changes in the physical asset, 4) composition of several different models, each of which could be used to yield certain information of the asset, and 5) responsiveness to changes in the asset [18].

After the physical twins, the next step in the development towards digital twins were intelligent products and software agents [19, 20, 21, 22], which emerged at the turn of the 21st century. According to Wong et al.
[20] an intelligent product has a unique identity, can communicate with its environment, is able to store the product data, can communicate its features to others, and has decision-making capabilities. Software agents were needed to implement the last two of these features and Wong et al. defined a software agent as a software process being able to reason, react to changes, and collaborate with other agents. Främling et al. [22] proposed that "each product item has a corresponding "virtual counterpart" or agent associated with it", and these "agents provide services for their physical counterparts." Främling et al. also introduced other concepts closely related to digital twin, such as link between physical product and its information, and making this information accessible from a single point on the Internet. Related to information management, Hribernik et al. [23] presented a product-centric model, in which the product links the actors during its life cycle (Figure 2.1), and introduced Product Avatar concept. Product Avatar is a virtual counterpart of a product that is an autonomous entity comparable to the concept of a software agent. It provides access to product information and can interact with other Avatars. Liu et al. [8] considered Product Avatar concept similar to the digital twin.

![Figure 2.1](image)

**Figure 2.1.** Hribernik et al. [23] proposed a product-centric approach (A) for information exchange between actors. In the traditional approach, each actor communicated directly with each other (B).

Even though the concept of software agents presented by Främling et al. [22] can be considered as the first published description of digital twins, Grieves is often kept as the founder of the digital twin concept in scientific literature [8] with his presentation about "Conceptual Ideal for PLM" (Product Lifecycle Management) in 2002\(^2\) [4]. Grieves' model included virtual space mirroring the physical space and a link between the spaces allowing bi-directional data flow [24]. It has also been argued that Grieves informally used the term "digital twin" as early as in the year 2000 [13].

\(^2\)There seems to be confusion in the reviewed literature, whether Grieves introduced the digital twin concept in 2002 as a part of his presentation to industry or in 2003 as a part of a PLM course. This is because the former claim is presented by Grieves and Vickers in [24] and the latter in his earlier white paper about digital twins [25].
The first publication that mentions "digital twin" is [26] by Hernández and Hernández from the year 1997. This publication referred "digital twin" as the 3D model of the built environment. In 2005, Nicolai, Resatsch and Michelis used "digital twin" in their figures [27], and, in 2010, Puig and Duran called a person’s avatar on the Internet a "digital twin" [28]. Nevertheless, it is widely acknowledged in the scientific community that the term "digital twin" was coined in NASA’s draft roadmap [29], which was published a few months after Puig and Duran’s conference.

NASA’s roadmap defined digital twin as follows [29]: "an integrated multi-physics, multi-scale, probabilistic simulation of a vehicle or system that uses the best available physical models, sensor updates, fleet history, etc., to mirror the life of its flying twin. The digital twin is ultra-realistic and may consider one or more important and interdependent vehicle systems...”

This first definition of digital twin focused on the aerospace field and accurate mirroring of the vehicle [30], but the focus was quickly expanded: Lee et al. [31] brought digital twin to the manufacturing environment as a coupled model of a real machine in 2013 and Ríos et al. [32] were the first to reference digital twin as a counterpart of generic "product" [30] in a scientific publication in 2015. Nevertheless, Grieves [25] had already presented digital twin as a counterpart of any product in his white paper published in 2014. What these early digital twins have in common is that they strive to describe the product accurately, "from the micro atomic level to the macro geometrical level" [24].

Gartner chose digital twin to its "Top 10 Strategic Technology Trends for 2017" [33], and, since 2017, the number of publications about digital twin has exploded [8]. Yet, no widely accepted definition nor methods to create a digital twin exist [8]. On the contrary, as the number of publications has risen, also the variety of digital twin definitions has increased. It is also acknowledged in the scientific community that there can be several views to the concept, such as modeling-oriented and information-management-oriented views [8], the latter of which is adopted in this thesis. Current digital twin definitions are very inclusive, such as the definition by Autiosalo et al. [6]: "a virtual entity that is linked to a real-world entity", to cover the high variety of digital twins. Therefore, the meaning and purpose of a digital twin is needed to be defined within each use case.

Digital twin is not the only initiative to bring digitalization into the industry. For example, Cyber-Physical Systems (CPS) and the Asset Administration Shell (AAS) are also proposed as enablers for Industry 4.0. Cyber-Physical systems combine the physical and digital world, and a digital twin is described as a cyber-part of the CPS [34]. AAS describes the properties of the device and provides an external interface to the device [35]. The third relevant technology related to the digital twin is OPC UA, a de facto industrial standard. This dissertation uses OPC UA mainly as a method to communicate with a machine PLC system. Nevertheless,
OPC UA may also provide some useful features for digital twins, such as a standardized information model and access to historical data [36].

2.2 Digital twin architectures

This section investigates the state of the art of digital twin architectures. The focus is on general architectures that must include sufficient details for implementing digital twin. Publications only concentrating on a single use case or one part of the digital twin, such as a data processing pipeline, are left out of this state-of-the-art review. In addition, the review concentrates on information-oriented digital twins suitable for a single industrial machine, and architectures describing factory-level digital twins are out of scope. The research questions for the state-of-the-art review are as follows:

1. Are there modular digital twin architectures that are suitable for various use cases?
2. Do the current architectures make the data available and accessible?
3. Are sufficient details provided to implement a digital twin following the architecture?
4. Are the architectures tied to a certain technology stack (for example, use of OPC UA)?
5. Can existing systems be utilized for building a digital twin?

The review was conducted using Scopus database and targeting “digital twin” and “architecture” to the title, abstract, and keywords. In addition, most relevant papers using microservices with digital twins are introduced in this section. Only articles in English were considered.

A Cloud-based Cyber-Physical System (C2PS) architecture was introduced by Alam and Saddik [37]. Entities can communicate either on the physical layer or via cyber-layer using peer-to-peer communication with the communication groups model. The architecture loosely follows Service-oriented Architecture (SoA) because each entity provides its capabilities as services. Digital twins can be constructed hierarchically consisting of sub-digital twins. C2PS is a high-level architecture and lacks implementation details.

Bazaz, Lohtander, and Varis [38] build upon the C2PS architecture and combined it with 5-layer digital twin architecture. The layers are the data store layer, processing layer, model and algorithms layer, analysis layer, and user interface layer. In addition, relevant technologies to implement digital twin, such as OPC UA, were considered. The architecture was demonstrated with a digital twin of a CNC (Computer Numerical Control) machine.
Park, Easwaran, and Andalam [39] presented Twin-in-the-Loop Architecture (TiLA), which is based on the Globally Asynchronous Locally Synchronous (GALS) model of computation (MoC). In this model, computational models can be synchronised locally with each other, but all models are not necessarily in sync. They used Twin Description File (TDF), which specifies the models used to create a digital twin, connections between models, and mapping of digital twin data to the data model. The developed digital twin offers an API, which can be used to access TDF and digital twin data. In addition, events can be subscribed from the API.

A digital twin architecture based on data, models, and services (DMS) framework was developed by Jiang, Guo, and Wang [40]. The digital twin is divided into local and cloud part. The local part is synced with the physical part in real-time, and it sends preprocessed data into the cloud. Computationally heavier analyses are run in the cloud, and other applications can fetch information from the digital twin via Remote Procedure Calls (RPC). Both local and cloud systems are constructed from several components that can communicate with each other. Local systems use APIs for communication whereas remote procedure calls (RPC) and RabbitMQ is used with the communication of cloud components. The implementation of the architecture for gas-insulated switchgear was presented.

Dasbach et al. [41] examined the communication of a digital twin. They divided a digital twin into three parts: front-end providing interfaces, core making decisions and running simulations, and back-end acting as storage. Each digital twin provides services, and an address from which these services can be requested is stored in a cloud “phonebook”. The services can be consumed directly using peer-to-peer communication between the twins or via the cloud.

A framework for Smart Manufacturing digital twins based on Software-Defined Control was introduced by Qamsane et al. [42]. The factory has several different types of digital twins divided into different classes, such as topology, machine asset, and machine process class, each of which has its own purpose. Communication between these digital twins and digital twins and applications uses the publish-subscribe model.

Moyne et al. [43] proposed an object-oriented framework for creating digital twins. Digital twins are built from objects, and object-oriented paradigms such as inheritance can be used. However, there are still many issues that need to be overcome before the practical implementation of the framework, such as communication between components and other digital twins.

An architecture in which digital twin data is provided via Web services allowing the use of various data stores was introduced in [44] by Schroeder et al. In other words, a digital twin acts as an interface from which applications fetch data. The architecture was demonstrated by providing data for an augmented reality (AR) application.
Shahriar et al. [45] presented OpenDT framework, which allows publishing and discovering digital twins and querying their services. Each digital twin provides its functionalities as a set of services via API following Service-oriented Architecture. The services and digital twin metadata is described in a DT Descriptor XML (Extensible Markup Language) document. The limitation of the paper was that the digital twin architecture was not described in a detailed manner.

Open-source software for building a digital twin was considered by Damjanovic-Behrendt and Behrendt [46]. In addition, an architecture for digital twin consisting of managers for virtualization (including data, models, and services managers), interoperability, monitoring, decision-making, and simulations was presented. It was proposed that these managers should be implemented using a microservices architecture. These microservices constructing a digital twin were briefly described.

A four-layer architecture for digital twins was introduced by Malakuti et al. [47]. The architecture consists of information providers, such as databases, model providers, digital twin providers for managing digital twins and making the models discoverable, and the application layer consuming the services offered by the previous layers. In this architecture, the digital twin provides a single access point of the product data. The architecture was implemented for motor speed drivers, and it consisted of digital twin for the specific drive and its type. Microservices were used to implement the components for each layer.

Preuveneers, Joosen, and Ilie-Zudor [48] focused on preventing error propagation and improving the robustness of digital twins using feature toggles. Their digital twin architecture consisted of data processing microservices and connections between these services. Composing digital twins from microservices enabled adding new features to twins easily. Microservices can be toggled off in runtime when errors are detected to prevent error propagation.

Mena et al. [49] proposed a Digital Dice concept utilizing Web of Things (WoT) Thing Description (TD) document. Digital Dices can be generated based on this document that describes the services offered by the dice. These services are implemented using microservices. The weakness of the paper is that the suitability for industrial machines is not considered, and it is not described how microservices communicate.

A six-layer architecture for a Cyber-Physical Production Systems focusing on a single manufacturing cell was presented by Redelinghuys, Basson, and Kruger [50]. Layers 1 and 2 consist of physical devices and their controllers, respectively. Layer 3 contains data repositories, including local databases and OPC UA servers communicating with the controllers. Layer 4 selects, fetches, and converts data from the data repositories in layer 3 to cloud information repositories in layer 5. It may also provide GUI as it has access to both layers 3 and 5. The current state and historical data of the
manufacturing cell are stored in layer 5. Layer 6 implements digital twin features and "intelligence". Because these features are use case specific, the implementation of layer 6 is left open. Nevertheless, Tecnomatix Plant Simulation has been given as an example of a commercial tool providing simulation capabilities. This architecture mainly concentrated on data exchange between the physical and digital world.

The results of the state-of-the-art review of digital twins are summarized in Table 2.1. The columns are based on literature review research questions, and were assessed on a three-step scale (yes, partly, no). Partly means either that the evaluation criteria were partly fulfilled or that the "feature" could be added in the future. For example, if data is stored in the cloud it can be made available, even though it was not considered in the original architecture. The evaluation criteria for the architectures were as follows:

**Modular** means that new features and components can easily be added to the digital twin and based on need.

**Flexible** describes suitability for different digital twins and use cases.

**Data availability** allows accessing the data by other systems.

**Implementation details** describe if sufficient details are provided to implement the architecture and if an example implementation is presented.

**Technology-lock** means that the architecture is bound to a certain technology or software, such as OPC UA or Tecnomatix Plant Simulation.

**Existing systems** describe if it is possible to use existing systems for implementing digital twin.

In conclusion, there are several architectures that are modular and flexible. Most of the solutions also bring the digital twin data available, but not necessarily from a single access point. The need for describing the digital twin and its features were identified in the reviewed literature. The proposed solutions for that included Twin Description File [39], "phonebook" [41], DT Descriptor document [45], and WoT TD [49]. The weaknesses of the current architectures are mainly related to the lack of implementation details and not using the existing systems. The lack of implementation details is surprising since many of the architectures presented an example implementation of a digital twin. However, these example implementations did not provide sufficient details for building a digital twin. Only two research papers had sufficient implementation details, [44] and [47]. These architectures also fulfilled the other criteria of the review. They are information-oriented and provide access to digital twin data via APIs. Thus, they are quite similar to the architecture presented in this work.
Literature review

Table 2.1. Summary of the state-of-the-art review of digital twin architectures.

<table>
<thead>
<tr>
<th>Ref.</th>
<th>Modular</th>
<th>Flexible</th>
<th>Data avail.</th>
<th>Impl. details</th>
<th>Tech-lock</th>
<th>Exist. systems</th>
</tr>
</thead>
<tbody>
<tr>
<td>[37]</td>
<td>Yes</td>
<td>Yes</td>
<td>Partly</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>[38]</td>
<td>Partly</td>
<td>Partly</td>
<td>Partly</td>
<td>Partly, ex. impl.</td>
<td>No</td>
<td>Partly</td>
</tr>
<tr>
<td>[39]</td>
<td>Yes</td>
<td>Partly</td>
<td>Yes</td>
<td>Partly, ex. impl.</td>
<td>No</td>
<td>Partly</td>
</tr>
<tr>
<td>[40]</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes, ex. impl.</td>
<td>No</td>
<td>Partly</td>
</tr>
<tr>
<td>[41]</td>
<td>Partly</td>
<td>Partly</td>
<td>Yes</td>
<td>Partly, ex. impl.</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>[42]</td>
<td>Yes</td>
<td>Partly</td>
<td>Partly</td>
<td>Partly, ex. impl.</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>[43]</td>
<td>Yes</td>
<td>-*</td>
<td>-*</td>
<td>No</td>
<td>-*</td>
<td>-*</td>
</tr>
<tr>
<td>[44]</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes, ex. impl.</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>[45]</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Partly, ex. impl.</td>
<td>No</td>
<td>Partly</td>
</tr>
<tr>
<td>[46]</td>
<td>Yes</td>
<td>Yes</td>
<td>Partly</td>
<td>Partly, ex. impl.</td>
<td>No</td>
<td>Partly</td>
</tr>
<tr>
<td>[47]</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes, ex. impl.</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>[48]</td>
<td>Yes</td>
<td>Partly</td>
<td>Partly</td>
<td>No</td>
<td>Partly</td>
<td></td>
</tr>
<tr>
<td>[49]</td>
<td>Partly</td>
<td>Partly</td>
<td>Yes</td>
<td>Partly, ex. impl.</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>[50]</td>
<td>Partly</td>
<td>Yes</td>
<td>Partly</td>
<td>Partly, ex. impl.</td>
<td>No</td>
<td>Partly</td>
</tr>
</tbody>
</table>

*Cannot be assessed because implementation details are missing.

Nevertheless, this work also compared two API design paradigms, REST and GraphQL, in industrial use and presented means to collect data from physical devices with OSEMA.

2.3 Research gap

There are no widely accepted methods and tools to build digital twins in practice, which hampers companies’ willingness to adopt digital twins [51]. In addition, the wide variety of proposed solutions combined with the absence of standardization impedes building upon the previous work [8]. These findings were also confirmed by the state-of-the-art review conducted in the previous section. Another major issue with digital twin architectures is the lack of implementation details, which makes the practical implementation of digital twin challenging. Even though there are a few promising general digital twin architectures identified in the state-of-the-art review, the following weaknesses of the current solutions were identified:

1. Not using the expertise of all stakeholders around a machine, such as the manufacturer, operator, and software provider. This may happen if the digital twin development focuses merely on software implementation.

2. Boundaries between stakeholders are unclear when implementing digital twin: who is responsible for each feature and who provides data.

3. Not adopting existing systems. It is time-consuming, inefficient, and
expensive to build a holistic digital twin from scratch.

4. Lack of scalability. Scalability would allow updating, replacing, or replicating the individual components of the digital twin based on need.

5. Digital twin data is not available from a single access point. Providing digital twin data from a single API would allow easier data access and access management that enables control to who can access and what data.

This dissertation presents a general API-based digital twin architecture adopting microservices, which addresses the aforementioned weaknesses of the current general digital twin architectures.
3. **Materials and methods**

This chapter presents the methods for creating and implementing an API-based architecture for digital twins, conducting comparison between REST and GraphQL, and developing OSEMA. First, the microservices architecture, which is the underlying model for API-based digital twin, is introduced. Thereafter, the used software, programming languages, and Web frameworks are presented. Finally, this chapter introduces the developed algorithms and the experimental setup used in Publications II and III.

3.1 **Microservices architecture**

In a microservices architecture, the system is composed of separate services, whereas a traditional monolithic system implements functionalities inside a single process (Figure 3.1 A and B) [52]. There is no strict nor standard definition for the microservices architecture [53], and the architecture is rather loose guidelines for creating scalable applications. The services composing a system communicate via messages, often using REST APIs [53]. Compared to monolithic systems, each service can be replicated individually, which enables scaling only the necessary parts of the application (Figure 3.1 C and D). For a more detailed view on microservices and their characteristics, history, and enabling technologies, see for example [53, 54].

Microservices architecture tries to solve limitations of monolithic systems. Microservices tend to promote clearer boundaries between modules, which helps to share responsibilities between teams and leads to enhanced modularity [55]. Modularity improves the maintainability of the system since each component can be updated separately, and there are fewer connections between services [53]. In addition, a breakdown of a single service does not cause the failure of the entire system [56]. Modular services also allow scaling the system more efficiently since only the needed services can be replicated [52] (Figure 3.1 C and D). Because each service can
Materials and methods

Figure 3.1. A monolithic system implements all functionalities in a single executable (A). In a microservices architecture, the functionalities of the system are implemented by separate services (B). Monolithic systems are scaled by replicating the whole system (C), whereas microservices architecture allows replicating only the necessary services (D).

be deployed independently, there is no need to reboot the whole system when functionalities are updated [55]. Microservices architecture allows choosing the used technologies freely and prevents lock-in to a certain technology stack [53].

Microservices architecture also has its own weaknesses, which are mainly related to the distributed nature of the system [55]. First of all, the performance is lower compared to monolithic systems because remote service requests over a network are slower than in-memory function calls [53]. Communication over a network also poses other issues, such as the reliability of communication [55] and overhead to enable secure communication [53]. Because the delivery of the message cannot be trusted or service might be unavailable at any time, microservices are not considered as
Materials and methods

reliable as monolithic systems [52, 53]. For this reason, microservices need to be designed resilient for failures [55].

In the microservices architecture, there is no centralized database, but every service manages its own storage [52]. This causes issues with the data consistency across the services [55]. Because data is also sent outside of the system and some services might be provided by third parties, whose data storing mechanisms might be vulnerable to attacks, the security needs to be considered more carefully than in monolithic systems [53]. A single service is simple within microservices, but the complexity of the system might translate into the relations between the services [55]. In addition, testing a single service is easy and straightforward, but testing the whole system is laborious because of several interconnections [53].

Microservices have been proposed as an enabling technology for digital twins [6, 14], and several publications have applied microservices on digital twins [47, 48, 49, 57, 58], the most relevant of which are presented as a part of digital twin architectures in Section 2.2. This dissertation brings the benefits of microservices architecture to digital twins. The microservices architecture is employed by implementing each digital twin feature as a separate service.

3.2 Digital Twin Document

During this research, it was noticed that a standardized way to describe the metadata of a digital twin is necessary. For that purpose, the development of a digital twin document (DT document) was initiated. DT document is a text-formatted human-readable file that allows examination of the digital twin metadata. Others have also identified a need for this kind of document, as found out in the literature review in Section 2.2. For example, Shahriar et al. [45] presented DT Descriptor document, Microsoft is developing Digital Twin Definition Language (DTDL) [59], and World Wide Web Consortium (W3C) has published WoT TD document [60]. The development of DT document does not aim to create yet another standard but communicates the elements that should be included in ongoing standardization efforts. The DT document standard draft could be merged as a part of DTDL or WoT TD in the future.

From a technical perspective, the DT document is a YAML file (YAML Ain’t Markup Language) with standardized fields. The compulsory fields are the version of the DT document standard, the classification of the document (public, private, or shared), the digital twin id, name, and description, and the time the digital twin was created. The optional parts of the document include, among others, other metadata such as location and owner, descriptions of the digital twin features, and relations to other digital twins. Figure 3.2 shows an example of the document’s public part for a
simple IoT device, a lamp in the fridge. The current version of the standard
draft can be found on GitHub (https://github.com/AaltoIIC/dt-document).

```
version: "1.0"
privacy: "public"
id: "http://lampthing.fi/lamp-l98765"
name: "Lamp in the fridge"
description: "Lamp installed in the fridge used to light up the refrigerator."
createdMachine: "1583195640"
createdHuman: "2020-03-02 19:34:00"
manufacturer: "Philips"
features:
  - name: "Sensor reading"
    description: "Sensor reading allows managing sensors attached to the lamp."
    address: "https://lampthing.fi/sensors/browse"
    requirement: "User account is needed."
    keywords:
      - "sensors"
      - "measurement"
      - "temperature"
      - "management"
```

**Figure 3.2.** DT document is a development effort to describe the metadata of a digital
twin in a standardized way. This is a screenshot from the GitHub page of the
standard draft (https://github.com/AaltoIIC/dt-document) showing an example
of the document.

Publication I also presented an idea that DT documents could be used
to create a global network of digital twins, analogously to the World Wide
Web (WWW). This network of digital twins would be called Digital Twin
Web (DTW). The concept and implementation of DTW with Twinbase was
introduced in [61] by Autiosalo, Siegel, and Tammi. DTW would allow
discovering digital twins and, in the future, means for digital twins to
interact with each other. Another purpose for the DTW is to provide
information about digital twins in an easy-to-access and user-friendly way.
Further examination of the DTW falls out of the scope of this dissertation.

### 3.3 Hardware

This dissertation focuses mainly on software development. Therefore, the
used hardware is mainly related to deploying software and servers. For the
deployment, primarily different versions of Raspberry Pi card computers
were used. In addition, Ilmatar overhead crane (Figure 3.3) and its OPC
UA server connected to the crane PLC were the main hardware of this work.
Ilmatar crane is not a standard crane from its manufacturer, and a more
detailed description can be found in [9]. Ilmatar is equipped with several
smart features, such as automated positioning and anti-sway control.

With OSEMA, Espressif ESP32-based microcontrollers manufactured
by Pycom were used (Figure 3.4). These microcontrollers were chosen
Materials and methods

because they support a wide variety of networks, such as LoRa (Long Range), NB-IoT (Narrowband Internet of Things), and SigFox [62]. In addition, developing was easy for the author since microcontrollers could be programmed with Micropython that is similar to Python programming language. Microcontrollers were used with expansion boards offering a USB connection, a MicroSD slot, and socket headers for wires. Sensors were connected to the microcontrollers via a low-power I2C bus [63], and the following sensors were tested with OSEMA: Analog Devices ADXL345 [64] (also visible in Figure 3.4) and STMicroelectronics LIS3DSH [65] three-axis accelerometers, an analog-to-digital converter, and Garmin LIDAR-Lite v3HP distance sensor [66].

Figure 3.3. Ilmatar overhead crane that is located at Aalto Industrial Internet Campus.

Figure 3.4. OSEMA used Pycom microcontrollers along with I2C bus sensors.
3.4 Technology stack

Figure 3.5 presents the technologies used in this thesis, including protocols, databases, and Web programming frameworks. It also gives an overview of what was implemented in practice; white color indicates that a block was implemented as a part of this dissertation, whereas blocks marked with red, that is, GraphQL wrapper and other digital twin features (and crane PLC with OPC UA server), were not implemented as a part of this work. Next, each of the technologies is introduced in more detail.

3.4.1 Protocols

HTTP was selected as a primary communication protocol because it is the most widely used protocol in the Internet, allowing maximal interoperability. Both REST and GraphQL interfaces are used with HTTP in practice, even though they can use any other reliable transport method (Publication II). HTTP is a request-response protocol and runs on top of TCP (Transmission Control Protocol) that provides reliable data transmission [67]. HTTP relies on a text-formatted header, and the overhead and resource consumption are relatively large compared to IoT protocols [68]. Secure communication with HTTP can be achieved by using TLS (Transport Layer Security) [69].

Sensor nodes connected to OSEMA fetch their software updates via HTTP but are able to send measurement data using MQTT (Message Queuing Telemetry Transport). MQTT is a lightweight publish-subscribe protocol for constrained devices and applications [70]. The publish-subscribe model is implemented with a broker that distributes messages from publishers to subscribers based on topics [71]. MQTT uses TCP at the transport layer to ensure high quality of service [72]. Compared to HTTP, MQTT requires lower computing power, has a smaller overhead, and uses less bandwidth [68].

Figure 3.5. Technologies used in this dissertation. White color indicates that the system was implemented as a part of this thesis, and red that it was not.
3.4.2 Web Development

This dissertation used three different Web frameworks: Django for OSEMA, Flask for the API gateway, and React and Node.js Express framework for the user interface of Data Link. Django is a scalable Python Web framework that provides a comprehensive set of built-in features, such as user-authentication and administrator page [73]. Therefore, developing applications is straightforward, and Django was a natural choice for OSEMA, which is a rather complex application. In addition, Django takes care of many security-related tasks, aiding the developer to create secure applications [73]. Flask [74] is also a popular Python-based Web framework, but compared to Django, it provides only a minimal set of included features. Thus, Flask is often described as a micro Web framework. The benefits of Flask include simplicity and fast deployment of applications with only a few lines of code. Because the API gateway is a rather simple application, it was chosen to be implemented with Flask. With both Django and Flask, a simple SQLite database was used.

The user interface of Data Link was programmed with JavaScript and used Node.js runtime, which is intended for the creation of scalable Web applications [75]. The front-end relied on React JavaScript library that allows building interactive user interfaces [76], and the back-end was implemented with Express Web framework [77]. This stack reflects the current trend in Web development, in which JavaScript is used both in the front and back-end. The user interface relied on MongoDB, which is a document database. This allows storing the DT document as it is in the database.

3.5 Algorithms

This section presents the most important algorithms used in this thesis. These algorithms include message forwarding for the API gateway, software generation and update process for sensor nodes, and how a sensor node reads data and sends it to the data server.

3.5.1 API gateway message forwarding

The API gateway gathers several APIs behind a single interface. This interface can then be used to access these APIs. The gateway forwards the incoming HTTP messages to the APIs and routes back their responses. The operation principle of the message forwarding algorithm is as follows:

---

\(^3\)The author acknowledges that the trends in Web development change often, and this technology stack might already be outdated when this dissertation is published.
1. Client requests a resource over HTTP from the gateway. The request is sent to a path that defines the service from which the resource is requested and its location, for example gateway.com/services/osema/sensors/1 (gateway URL (Uniform Resource Locator) /services/<service name>/<subpath i.e. resource location>)

2. The gateway validates the authentication token sent along with the request. Invalid token leads to 401 Unauthorized response. If the token is valid, but the user does not have the right to access this service or resource, the gateway responds with 403 Forbidden.

3. The gateway fetches information necessary to connect to the service from its database. This information includes the URL of the service, the authentication method, and credentials. The service is searched from the database with the name given in the address, and, if the service is not found, the API gateway returns 404 Not Found status.

4. The authentication-related fields in the original request are replaced with the information fetched from the database. In addition, the target URL field of the request is changed from the gateway URL to the target service URL.

5. The request is sent to the target service.

6. Depending on the response from the service, the gateway either:
   
a) forwards message back to the client, and message forwarding is completed.
   
b) tries to refresh the authorization token/credentials for the target service if the response code was 401 Unauthorized. In the case refresh is successful, the gateway resends the request with updated authorization headers and returns a response to the client. Otherwise, the gateway returns the initial response to the client.

### 3.5.2 Generating software for a sensor node

The rest of the algorithms are related to OSEMA. Figure 3.6 gives an overview of the algorithms and illustrates interactions between a sensor node, OSEMA, and the user. Both measurement and software update algorithms are run on separate threads so that they can be run continuously and independently of each other.

The software generation process starts when a new sensor node is added to OSEMA and initial software for that is needed, or when the configuration of an existing node is modified. The step-wise description of the software generation algorithm is as follows:
1. The sensor node configuration is fetched from the database of OSEMA. The configuration includes sensor model, measurement parameters, data transmission settings, network settings, and security-related settings. Table 3.1 lists and describes all the configurable parameters. Steps 3, 4, 5, and 7 use the configuration to select what is written to the file. For the selection, simple if-(elif)-else clauses are used.

2. A new file is created for the node software and opened for writing. The filename includes the sensor id and the timestamp of the creation.

3. The necessary library imports are written to the beginning of the file.

4. The libraries that are not included in the standard MicroPython package are written directly to the file because the node software needs to be on a single file.

5. The global variables are written to the file. These include, for example, settings for security, such as shared secret key for software updates, measurement parameters, I2C address-value pairs for configuring the sensor settings, and I2C register address from which measurement data is read.

6. The functions that are always the same regardless of the node configuration are written to the file.

7. The functions that are configuration-specific are written to the file.
8. The file is closed and software generation is finished. OSEMA waits for a node to ask for updates or a user to download the file.

### 3.5.3 Sensor node software update process

Software updates are initiated by a sensor node since there is no method to push data from OSEMA to sensor nodes. A node asks for software updates from OSEMA with the predefined interval, which can be freely chosen. The software update request is sent either over HTTP or HTTPS. The request includes the node software version and a nonce. The request is encrypted with secure [78] AES (Advanced Encryption Standard) 128-bit CBC (Cipher Block Chaining) to ensure confidentiality, and SHA-256 (Secure Hash Algorithm) HMAC (keyed-Hash Message Authentication Code) with a 128-bit key is used to provide authenticity and integrity. These methods are also used to secure responses from OSEMA. To prevent replay attacks, the response from OSEMA must contain the nonce sent along with the request. A nonce is a random 128-bit key that allows a client to check if a response matches a certain request.

When a new software update request is received by the server, the integrity and authenticity of the message are checked. This is conducted by hashing the message using the same key as the sensor and comparing this calculated HMAC with the HMAC sent along with the message. After that, the message is decrypted with the shared secret key that is unique for each sensor node. The software version sent along with the response is compared to the version in the OSEMA database. If OSEMA has a newer version, it sends a response that contains a new software. Otherwise, OSEMA responds that the software is up to date and changes the sensor status to up to date. When a sensor node receives a response, it similarly checks the HMAC of the message and decrypts the message. If the sensor node receives software up to date message, it continues the normal operation and asks for an update again after the initially specified interval. If it receives a new software, the node stores the software to its flash memory. Thereafter, it reboots, and during the boot phase, it removes the old software and renames the new software to `main.py`. The node runs the new software after start-up and immediately asks for updates from OSEMA. If updates are not available, the sensor node continues normal operation and OSEMA changes the sensor status up to date.

### 3.5.4 Reading data from a sensor

After boot-up, the sensor node starts a new thread for reading data from an I2C sensor. The actual measurement function is then called periodically, and the period time is defined by the sample rate. Periodical function calls yield more consistent measuring frequency when the frequency is high.
### Table 3.1. Configurable settings of sensor nodes (Publication III).

<table>
<thead>
<tr>
<th>Setting</th>
<th>Description</th>
<th>Example Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Update URL</td>
<td>OSEMA server URL where updates are requested</td>
<td><a href="http://www.example.com">www.example.com</a></td>
</tr>
<tr>
<td>Update port</td>
<td>Sensor node requests updates from this port</td>
<td>443</td>
</tr>
<tr>
<td>Update over HTTPS</td>
<td>Use HTTPS for sending updates (true/false)</td>
<td>True</td>
</tr>
<tr>
<td>Update check limit</td>
<td>How often a node requests updates</td>
<td>3600 (s)</td>
</tr>
<tr>
<td>Sensor model</td>
<td>Which sensor model is used</td>
<td>ADXL345</td>
</tr>
<tr>
<td>Sample rate</td>
<td>Sample rate of the sensor</td>
<td>12.5 (Hz)</td>
</tr>
<tr>
<td>Sensitivity</td>
<td>Sensitivity of sensor</td>
<td>± 2g</td>
</tr>
<tr>
<td>Burst length</td>
<td>Length of measurement when measured in bursts</td>
<td>10 (s)</td>
</tr>
<tr>
<td>Burst rate</td>
<td>Time interval between bursts</td>
<td>10 (s)</td>
</tr>
<tr>
<td>Data send rate</td>
<td>How often data are sent to the data server</td>
<td>10 (s)</td>
</tr>
<tr>
<td>Connection close limit</td>
<td>If the data send rate is higher than this, the</td>
<td>3 (s)</td>
</tr>
<tr>
<td></td>
<td>connection is closed after data transmission</td>
<td></td>
</tr>
<tr>
<td>Network close limit</td>
<td>If the data send rate are higher than this, the</td>
<td>30 (s)</td>
</tr>
<tr>
<td></td>
<td>network connection is closed after data</td>
<td></td>
</tr>
<tr>
<td></td>
<td>transmission</td>
<td></td>
</tr>
<tr>
<td>SSID</td>
<td>SSID of the Wi-Fi network</td>
<td>myHotspot</td>
</tr>
<tr>
<td>Security</td>
<td>The security method of the Wi-Fi network</td>
<td>WPA2</td>
</tr>
<tr>
<td>Key</td>
<td>Password of the Wi-Fi network</td>
<td>secretpasswd</td>
</tr>
<tr>
<td>Username</td>
<td>Username for Wi-Fi network</td>
<td>username</td>
</tr>
<tr>
<td>Protocol</td>
<td>Application layer protocols used for measurement</td>
<td>MQTT</td>
</tr>
<tr>
<td>Data server URL</td>
<td>Data server URL</td>
<td><a href="http://www.example.com">www.example.com</a></td>
</tr>
<tr>
<td>Data server port</td>
<td>Data server port</td>
<td>80</td>
</tr>
<tr>
<td>Path</td>
<td>Path where the measurement data are sent (with</td>
<td>/add-data</td>
</tr>
<tr>
<td></td>
<td>HTTP/HTTPS</td>
<td></td>
</tr>
<tr>
<td>MQTT User</td>
<td>Username for MQTT</td>
<td>username</td>
</tr>
<tr>
<td>MQTT Key</td>
<td>Password for MQTT</td>
<td>secretpasswd</td>
</tr>
<tr>
<td>Topic</td>
<td>MQTT topic</td>
<td>example/topic</td>
</tr>
<tr>
<td>Broker URL</td>
<td>MQTT broker URL</td>
<td>io.adafruit.com</td>
</tr>
<tr>
<td>Broker port</td>
<td>MQTT broker port</td>
<td>1883</td>
</tr>
<tr>
<td>Data format</td>
<td>In which format measurement data are sent</td>
<td>JSON</td>
</tr>
<tr>
<td>Variable name</td>
<td>Each variable should be named</td>
<td>x_acceleration</td>
</tr>
<tr>
<td>Encrypt data</td>
<td>Encrypt the measurement data (true/false)</td>
<td>True</td>
</tr>
<tr>
<td>Key for data encryption</td>
<td>Key used to encrypt data as hex number</td>
<td>3ba19f5c4192b131</td>
</tr>
<tr>
<td></td>
<td></td>
<td>7123e993ca9da21</td>
</tr>
</tbody>
</table>
Materials and methods

than using sleep commands in the measurement loop. The measurement data is read from the sensor register via I2C bus, and these data are currently saved to the flash memory of the microcontroller. As the flash memory size is only 4 MB, the number of measurements is fairly limited before data is needed to be sent to the data server. Nevertheless, the storage size could easily be increased with a microSD card in the future.

The node configuration specifies the interval in which measurement data is sent to the data server. By multiplying this interval with the sample rate, the sensor node calculates the number of measurements before data transmission is triggered. When this number is reached, the node starts a new thread for the data transmission so that it does not interfere with the measurement process. The measurement data is translated to the desired format and, if needed, encrypted. Thereafter, the sensor node either sends data immediately or first connects to the network depending on the configuration. To save power, the sensor node can be configured to turn off network connection between data transmissions. Finally, the measurement data is sent over HTTP(S) or MQTT to the data server.

3.6 Measurement setup for analyzing network traffic

In Publications I and II, the network traffic was captured to analyze processing times and the number of bytes sent. A measurement setup with a similar structure was used in both cases shown in Figure 3.7. All servers, clients, and interfaces were connected to the same router that mirrored the network traffic to one of the Ethernet ports. The traffic was then recorded from that port by using Wireshark software. Wireshark is the de facto tool for analyzing network traffic [79].

The hardware consisted of Raspberry Pi 4 card computers and laptops. In Publication I, the client, the API, and the API gateway were run on separate Raspberry Pi 4’s (whose RAM size varied), and traffic was captured by a laptop. In Publication II, client and REST/GraphQL API were run on separate laptops having 8 GB of RAM and Intel i5-7300U and i5-7200U processors, respectively. For the REST API, a platform employing Microsoft .NET Core and ASP.NET Core frameworks and programmed with C# was used. This platform was developed by Cavalieri et al. [80], and it is available from Github [81]. For the GraphQL API, the implementation by Hietala et al. [82] was used, which is also available from Github [83]. The GraphQL interface was slightly modified by adding JSON (JavaScript Object Notation) Web Token (JWT)-based authentication to have a similar authentication method with the REST API. In addition, because there were problems to get JWT-authentication working with the original Python-based web framework Starlette, the framework was
changed to FastAPI. The OPC UA server was run on Raspberry Pi 4 Model B 2 GB, and the traffic was captured by a laptop acting as a client.

Figure 3.7. The schematic of the measurement setup. The traffic between systems is mirrored to one of the router ports and Wireshark is used to capture the traffic.

4FastAPI (https://fastapi.tiangolo.com/) is a Python-based Web framework that uses Starlette underneath. Thus, the performance of the FastAPI-based implementation is comparable to the Starlette-based implementation.
4. Results and remarks

This chapter presents a general API-based architecture for digital twins. The architecture uses Data Link to bring digital twin features behind a single interface and enable their communication. In addition, this chapter compares Web APIs that are used to connect services to Data Link and introduces one digital twin feature, OSEMA, allowing data collection.

4.1 API-based Digital Twin Architecture

The main result of this dissertation is a general architecture for industrial digital twins answering the research problem: "How to build an industrial digital twin that is modular, makes digital twin data available, and uses existing systems?" This architecture is called API-based Digital Twin Architecture (ADTA), and it allows building digital twins from separate systems connecting them via Data Link. Next, the architecture is introduced in more detail, a practical implementation of Digital Twin following ADTA is presented, and the performance of the API gateway of Data Link is analyzed.

4.1.1 Architecture

The API-based Digital Twin Architecture is built around Data Link concept that was first presented by Autiosalo et al. [6]. Data Link has two main functionalities: 1) allow communication between systems implementing the features of digital twin and 2) provide a single interface to access digital twin features. These functionalities of Data Link answer to research questions 1: "How to build a digital twin in practice exploiting existing systems?" and 2: "How to make digital twin data and features available?", respectively. The interface provided by Data Link is used by external applications, user interfaces, and systems constructing digital twin as illustrated in Figure 4.1. Data Link also provides a user interface to examine the metadata and features of the digital twin, which are described
The prerequisite for building digital twins using Data Link is that each feature of the digital twin provides an API. Data Link then links these APIs behind a single interface using an API gateway. The API gateway forwards messages from the interface to the requested features, and the forwarding algorithm is presented in more detail in Section 3.5.1. The gateway enables monitoring of the network traffic and lowers the connections between systems. Fewer connections significantly reduce the work when connecting new systems to the digital twin. In addition, the gateway provides centralized management of the authentication. The authentication method for each system is stored on the gateway and, thus, each system needs to authenticate itself only to the gateway in order to access other systems.

The authentication process and the API gateway must be secure to prevent access to sensitive data and physical devices. The security of the gateway can be ensured using standard Web security practices such as the use of a secure communication channel provided by HTTPS. An extra threat to the security of the gateway is posed by the linked systems that may also be vulnerable to attacks. For example, a compromised system could use the gateway to attack other systems. Therefore, access rights should be given cautiously, and the network traffic via gateway should be constantly monitored to detect abnormal activities. Further considerations of API gateway security are not in the scope of this work.

ADTA is analogous to a microservices architecture as it divides digital
Results and remarks

twins into smaller software blocks that communicate via APIs. The division is based on features following the Feature-based Digital Twin framework by Autiosalo et al. [6]. There are already several stakeholders around an industrial machine, such as manufacturer, end-user (factory), and software provider. Dividing a digital twin into features helps to share responsibilities between these stakeholders. Each of them already has its own systems, for example, PLM, MES (Manufacturing Execution System), and maintenance system. To create a digital twin, these systems are needed to be connected, and this can be done using Data Link. Creating digital twin this way also prevents vendor-lock since each system can be replaced easily. In addition, scalability is improved since systems can individually be replicated, and new features can be introduced by adding new systems or services to the digital twin.

The user interface of Data Link gives information from the digital twin based on the DT document introduced in Section 3.2. This information includes descriptions of digital twin features and the metadata, such as location. Changes to the metadata can be made by directly modifying the DT document from the interface. The overview of the interface is shown in Figure 4.2. In addition, the user interface provides search functionality to find relevant digital twin features (Figure 4.3). The search is currently limited to the keywords of the features. The technologies used to build Data Link’s API gateway and user interface are presented in Section 3.4.2.

Figure 4.2. Screen capture of the Data Link user interface (Publication I). The user interface allows examination of digital twin metadata and features.
4.1.2 Implementation for Ilmatar crane

To test the suitability of ADTA for industrial machines, a digital twin was created for Ilmatar overhead crane. Ilmatar is a so-called smart crane as it has several advanced features such as anti-sway control and automated positioning. It is also necessary to note that Ilmatar is not a standard crane from the manufacturer. As being a smart crane, Ilmatar is already connected to several IT (Information Technology) systems, such as historical data storage and remote monitoring system. In addition, it offers an OPC UA server that provides access to the crane PLC system. The purpose of Data Link is to connect these currently separate systems to create a uniform digital twin that provides access to these systems via a single interface.

For connecting the features via Data Link, they need to have Web APIs. In the case of Ilmatar, two systems did not have APIs – Teamcenter and TRUCONNECT. For Teamcenter, the most relevant CAD files were hosted on a separate Web server that could be accessed via Data Link. For TRUCONNECT, Data Link provided basic information about the system and how it could be accessed. This method can also be applied in general to systems without APIs. The overview of the Ilmatar digital twin is presented in Figure 4.4. The systems that were connected to the Ilmatar digital twin were as follows:

1. Siemens MindSphere storing the crane historical data, such as hook position and load.
2. Crane OPC UA server allowing access to the crane PLC system. The OPC UA was connected to Data Link via GraphQL interface developed by Hietala et al. [82]. GraphQL and REST interfaces for OPC UA were compared in Publication II, and the results are examined in Section 4.2.

3. OSEMA managing the retrofitted sensors. OSEMA was developed as a part of Publication III, and it is presented in Section 4.3.

4. Remion Regatta providing advanced data analysis and data storage. Regatta was used to create an application estimating the usage roughness of the crane based on acceleration data from the hook. A more detailed description of the application can be found in Publication III.

5. Siemens Teamcenter PLM system storing CAD models of the crane. Teamcenter was not directly connected to Data Link, but the surface model of the crane in STEP and Siemens NX formats were saved on a Web server, which was then connected to Data Link.

6. Konecranes TRUCONNECT offering remote monitoring. Data Link contains only information on how to access this service.

![Ilmatar digital twin consisted of several separate systems](image-url)

**Figure 4.4.** Ilmatar digital twin consisted of several separate systems (Publication I).

### 4.1.3 Measurements

Measurements were conducted to examine the effect of the API gateway on the request execution times. The measurement setup is introduced in Section 3.6. The execution times were measured with two different POST requests and two different GET requests. Responses for POST requests
Results and remarks

Figure 4.5. Comparison of request execution times with and without the API gateway. The gateway adds a considerable amount of latency for the request execution times (Publication I).

Measurements clearly show that the gateway adds a considerable amount of latency (Figure 4.5). With small requests and responses, the difference is more than five-fold, and, with larger ones, the difference is still approximately two-fold. It is expected that with larger requests, the gateway roughly doubles the execution time because two round trips are needed instead of one. The latency could be minimized by starting to stream data immediately to the client when the first chunk from the target system arrives, instead of waiting for the whole request to be completed. The latency also depends on the locations of the services. Table 4.1 shows the statistical properties of the measurements, including standard deviation and minimum, maximum, mean, and median values.

Table 4.1. Request execution times in milliseconds for different requests and responses (Publication I). GW = API gateway, SD = standard deviation.

<table>
<thead>
<tr>
<th>Test</th>
<th>Min</th>
<th>Max</th>
<th>Mean</th>
<th>Median</th>
<th>SD</th>
</tr>
</thead>
<tbody>
<tr>
<td>GET small</td>
<td>6.8</td>
<td>8.4</td>
<td>7.1</td>
<td>7.1</td>
<td>0.33</td>
</tr>
<tr>
<td>GET small via GW</td>
<td>38.8</td>
<td>47.1</td>
<td>40.5</td>
<td>40.7</td>
<td>1.15</td>
</tr>
<tr>
<td>POST small</td>
<td>7.2</td>
<td>8.6</td>
<td>7.5</td>
<td>7.7</td>
<td>0.38</td>
</tr>
<tr>
<td>POST small via GW</td>
<td>36.4</td>
<td>47.8</td>
<td>41.8</td>
<td>42.0</td>
<td>1.37</td>
</tr>
<tr>
<td>GET large</td>
<td>94.2</td>
<td>115.6</td>
<td>106.2</td>
<td>108.1</td>
<td>5.31</td>
</tr>
<tr>
<td>GET large via GW</td>
<td>202.1</td>
<td>245.0</td>
<td>227.3</td>
<td>226.4</td>
<td>8.47</td>
</tr>
<tr>
<td>POST large</td>
<td>95.2</td>
<td>142.6</td>
<td>121.2</td>
<td>121.7</td>
<td>12.09</td>
</tr>
<tr>
<td>POST large via GW</td>
<td>178.6</td>
<td>285.3</td>
<td>244.4</td>
<td>244.5</td>
<td>18.47</td>
</tr>
</tbody>
</table>
4.2 Web interfaces for industrial communication

Publication II compared REST and GraphQL Web interfaces for OPC UA servers. The motivation for bringing Web interfaces to the industry is improved interoperability and usability. This section answers the research questions 3: "What are the differences between REST and GraphQL Web APIs in industrial communication?" and 4: "Which one is more suitable to create API for the OPC UA server, REST or GraphQL?" Regarding to research question 3, Table 4.2 summarizes the features of REST and GraphQL. OPC UA was added as a reference for the de facto industrial standard. Next, each feature presented in the table is assessed in more detail.

Table 4.2. Comparison between the REST and GraphQL interfaces (Publication II). A column for OPC UA was added as a reference for industrial protocol. The values are relative to each other, not absolute values.

<table>
<thead>
<tr>
<th>Feature</th>
<th>REST</th>
<th>GraphQL</th>
<th>OPC UA</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Communication model</strong></td>
<td>Client-server</td>
<td>Client-server and Subscriptions</td>
<td>Client-server and Publish-Subscribe [36]</td>
</tr>
<tr>
<td><strong>Protocol</strong></td>
<td>HTTP(^1)</td>
<td>HTTP(^1)</td>
<td>TCP/IP(^2)</td>
</tr>
<tr>
<td><strong>Cache</strong></td>
<td>At any point</td>
<td>App-specific(^3)</td>
<td>Server-side(^4)</td>
</tr>
<tr>
<td><strong>Scalability</strong></td>
<td>Good</td>
<td>Medium</td>
<td>Poor</td>
</tr>
<tr>
<td><strong>Interface</strong></td>
<td>Uniform</td>
<td>App-specific</td>
<td>Stand. services(^5)</td>
</tr>
<tr>
<td><strong>Ease of use/dev.</strong></td>
<td>Medium/Good</td>
<td>Good/Good(^6)</td>
<td>Poor/Medium(^7)</td>
</tr>
<tr>
<td><strong>BW usage</strong></td>
<td>High</td>
<td>Medium</td>
<td>Low</td>
</tr>
<tr>
<td><strong>Performance</strong></td>
<td>Low</td>
<td>Medium</td>
<td>Good</td>
</tr>
</tbody>
</table>

\(^1\) Not bound to a specific protocol, but used in practice with HTTP. \(^2\) Other protocols, such as WebSockets and, for publish-subscribe, AMQP and MQTT, can also be used [36]. \(^3\) Implementing cache is up to the application developer. \(^4\) A server can cache sensor readings, and the client can use the `maxAge` parameter to define the maximum age of the cached response [87]. \(^5\) OPC UA server offers a set of standardized services that the client consumes [36]. \(^6\) Allows a schema-first approach [86]. \(^7\) These are based on the author's experience.

Both REST and GraphQL use the client-server communication model [84, 85], in which a client requests services from the server. GraphQL also offers a possibility for subscriptions, in which the server can push value updates to client [86]. Subscription-like communication can also be achieved with REST by using HTTP long-polling, in which the client sends a request to which the server only responds when the requested resource is updated [88]. However, this violates the original design principles of REST. Subscriptions are more efficient than a client-server model if data are constantly changing, which is often the case in industrial communication, and data updates need to be sent to clients immediately.

HTTP is the most pervasive Web protocol, and therefore REST and
GraphQL use it for client-server communication [85]. However, neither are bound to the use of HTTP, and, for example, Web sockets that allow bi-directional communication are used to implement the subscriptions with GraphQL [86]. The difference between GraphQL and REST is that GraphQL uses HTTP only as a transport protocol, but REST also utilizes its methods (GET, POST, PUT, DELETE), which have semantic meaning.

REST allows easier caching since the URL used to access resources provides a unique identifier allowing HTTP caching [89]. The cache can be stored at any point in the route from the client to the server. GraphQL does not assign a unique identifier, which can be used for caching, by default [89], and the developer is responsible for implementing caching. Nevertheless, there exist tools that implement caching, such as Apollo Client that provides client-side caching [86]. The importance of caching depends on how quickly the data becomes obsolete, for example, caching cannot be used with sensor readings. However, when applicable, caching can improve performance considerably.

Caching also positively affects the server scalability, which is determined by the number of clients it can handle. As REST has more versatile caching, it should also provide better scalability than GraphQL. However, Heredia, Flores-García and Solano [90] conducted measurements indicating that GraphQL outperforms REST with a large number of clients in terms of request execution times. Scalability cannot be considered an important factor in industrial communication since the number of clients is expected to be relatively low.

REST provides the so-called uniform interface [84], defined by: standard operations on resources (read, create, update, and delete, that are linked to HTTP methods GET, POST, PUT, and DELETE); identification of resources with URLs; use of HTTP status codes to communicate server-side events; description of the response format with HTTP headers; content negotiation; and HATEOAS (Hypermedia as the Engine of Application State). On the other hand, GraphQL provides three types of operations: query for retrieving resources, mutation for modifications, and subscriptions to get updates from servers [86]. GraphQL allows nested queries to interact with multiple objects with a single query [91]. In addition, the user can define which data is returned [86], which along with nested queries, greatly reduces response sizes compared to REST. The GraphQL interface is application-specific, and it is defined in the schema that allows introspection of the API structure [91]. The application-specific interface is more efficient than the uniform interface [84], but the uniform interface has other benefits such as hyperlinks that allow referencing to resources. Because of the efficiency and introspectable schema, the application-specific interface of GraphQL can be considered a better choice for industrial use.

User tests by Brito and Valente [92] suggested that GraphQL is easier and faster to use than REST. The test users found the GraphiQL tool
and clearer syntax as the major advantages over REST. GraphiQL is a Web user interface that allows inspecting schema and aids in writing the queries \[86\] helping both users and developers. REST is a more established paradigm meaning that developers are more familiar with it, and there should be wider support of different tools. Ease of use and development are crucial for adopting new technologies in the industry and accelerating the Fourth industrial revolution.

The measurement setup and the implementations of interfaces for assessing the bandwidth usage and the performance of REST and GraphQL are introduced in Section 3.6. The measurements were conducted by reading and writing a single value either once or 50 times. Figure 4.6 shows the TCP payload with a single value and Figure 4.7 with 50 values. From these figures, it is clear that GraphQL consumes considerably less bandwidth than REST, and OPC UA binary is the most efficient way to communicate with an OPC UA server. GraphQL queries were also performed in a minified form, in which extra characters, such as spaces, are removed. The REST interface returns an unnecessary token as part of the response, and the tests were also run without this token by modifying the interface. It should also be noted that even though REST and GraphQL interfaces use similar JWT-based authentication, the length of the token is 126 bytes longer with REST, which affects the TCP payload.

![Figure 4.6. The payload sizes with one value (Publication II). min. = minified, w/o = without.](image)

REST is slightly faster than GraphQL when a single value is written and the interface is connected to the OPC UA server before request. In reading, there was no statistically significant difference. On the other hand, GraphQL performs significantly better when a value is read or written multiple times since it needs only a single request compared to REST, which sends one request per value. To examine the effect of multiple handshakes needed to connect to the OPC UA server, request execution times were also measured when the connection to the OPC UA server was first needed to be established. Even though the Web interfaces are already connected to the OPC UA server, they are still slower than using the OPC
Results and remarks

Figure 4.7. The payload sizes with 50 values (Publication II). min. = minified, w/o = without.

UA server directly because connection establishment was rather quick. It was also noticed that the REST interface was very slow at connecting because it sends unnecessary messages to the OPC UA server. The request execution times are summarized in Table 4.3, which shows that using OPC UA directly outperforms requests via additional interfaces.

In summary, GraphQL offers a more favorable set of features for industrial communication than REST. Especially GraphiQL tool, introspectable schema, nested queries, and ease of use are great advantages over REST. Therefore, it is preferred that systems are connected to Data Link via GraphQL API. In addition, GraphQL is more suitable for use with OPC UA server than REST. However, using OPC UA directly consumes less bandwidth and yields lower request execution times. Therefore, it is recommended to have GraphQL as an additional interface for an OPC UA server.

4.3 Data collection

Mirroring and reflecting the physical world has been identified as one of the key characteristics of a digital twin [4, 8]. For achieving data flow from the physical world to the virtual world, this dissertation developed OSEMA for easy addition and management of sensors. OSEMA contributes to the research questions 5: “How to collect data from an industrial machine for its digital twin?” and 6: "How to manage data collection and change sensors’ settings over the Internet?"

OSEMA is able to generate software for sensor nodes based on their configuration. This algorithm and configurable settings are described in Section 3.5.2. The software generation ability allows adding new nodes without manual programming and is the basis for remote software updates.
Table 4.3. The request execution times in milliseconds for reading and writing values to the OPC UA server. SD = standard deviation. Disconnected and connected indicate if the OPC UA session was established before the request was made.

<table>
<thead>
<tr>
<th></th>
<th>Min</th>
<th>Max</th>
<th>Mean</th>
<th>Median</th>
<th>SD</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Read 1 value</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Connected</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>OPC UA</td>
<td>4.3</td>
<td>7.6</td>
<td>5.6</td>
<td>5.9</td>
<td>0.8</td>
</tr>
<tr>
<td>REST</td>
<td>42.6</td>
<td>110.8</td>
<td>70.9</td>
<td>70.7</td>
<td>9.8</td>
</tr>
<tr>
<td>GraphQL</td>
<td>38.1</td>
<td>100.8</td>
<td>71.0</td>
<td>75.7</td>
<td>14.4</td>
</tr>
<tr>
<td><strong>Disconnected</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>OPC UA</td>
<td>24.1</td>
<td>38.1</td>
<td>31.6</td>
<td>32.7</td>
<td>4.1</td>
</tr>
<tr>
<td>REST</td>
<td>1360.3</td>
<td>1509.5</td>
<td>1432.3</td>
<td>1425.5</td>
<td>29.1</td>
</tr>
<tr>
<td>GraphQL</td>
<td>75.5</td>
<td>147.9</td>
<td>102.7</td>
<td>107.7</td>
<td>21.0</td>
</tr>
<tr>
<td><strong>Write 1 value</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Connected</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>OPC UA</td>
<td>3.9</td>
<td>6.0</td>
<td>4.7</td>
<td>4.6</td>
<td>0.5</td>
</tr>
<tr>
<td>REST</td>
<td>26.0</td>
<td>58.3</td>
<td>49.7</td>
<td>52.2</td>
<td>6.8</td>
</tr>
<tr>
<td>GraphQL</td>
<td>54.4</td>
<td>96.0</td>
<td>73.6</td>
<td>66.0</td>
<td>12.4</td>
</tr>
<tr>
<td><strong>Disconnected</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>OPC UA</td>
<td>22.9</td>
<td>139.6</td>
<td>28.4</td>
<td>25.5</td>
<td>16.0</td>
</tr>
<tr>
<td>REST</td>
<td>1359.8</td>
<td>1616.4</td>
<td>1409.6</td>
<td>1399.5</td>
<td>43.4</td>
</tr>
<tr>
<td>GraphQL</td>
<td>78.7</td>
<td>142.4</td>
<td>103.9</td>
<td>95.2</td>
<td>19.1</td>
</tr>
<tr>
<td><strong>Read 50 values</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Connected</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>OPC UA</td>
<td>15.2</td>
<td>43.5</td>
<td>17.8</td>
<td>16.3</td>
<td>5.4</td>
</tr>
<tr>
<td>REST</td>
<td>915.0</td>
<td>3042.8</td>
<td>1271.8</td>
<td>943.4</td>
<td>626.9</td>
</tr>
<tr>
<td>GraphQL</td>
<td>142.0</td>
<td>272.8</td>
<td>205.2</td>
<td>205.5</td>
<td>30.3</td>
</tr>
<tr>
<td><strong>Disconnected</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>OPC UA</td>
<td>49.0</td>
<td>90.1</td>
<td>60.3</td>
<td>57.2</td>
<td>7.2</td>
</tr>
<tr>
<td>REST</td>
<td>2329.9</td>
<td>4061.9</td>
<td>3088.9</td>
<td>3156.0</td>
<td>421.6</td>
</tr>
<tr>
<td>GraphQL</td>
<td>147.0</td>
<td>275.8</td>
<td>224.7</td>
<td>231.5</td>
<td>25.2</td>
</tr>
<tr>
<td><strong>Write 50 values</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Connected</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>OPC UA</td>
<td>20.0</td>
<td>53.1</td>
<td>25.3</td>
<td>24.1</td>
<td>6.1</td>
</tr>
<tr>
<td>REST</td>
<td>815.0</td>
<td>2643.5</td>
<td>1054.8</td>
<td>836.1</td>
<td>470.7</td>
</tr>
<tr>
<td>GraphQL</td>
<td>222.7</td>
<td>353.5</td>
<td>268.0</td>
<td>262.0</td>
<td>29.0</td>
</tr>
<tr>
<td><strong>Disconnected</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>OPC UA</td>
<td>58.5</td>
<td>89.1</td>
<td>70.5</td>
<td>69.6</td>
<td>6.6</td>
</tr>
<tr>
<td>REST</td>
<td>2139.5</td>
<td>3446.9</td>
<td>2868.6</td>
<td>2924.5</td>
<td>243.9</td>
</tr>
<tr>
<td>GraphQL</td>
<td>236.4</td>
<td>330.5</td>
<td>275.8</td>
<td>269.5</td>
<td>24.2</td>
</tr>
</tbody>
</table>
The secure software update process is introduced in Section 3.5.3.

Figure 4.8 shows the Browse sensors view that lists all sensor nodes connected to OSEMA. A sensor node can be modified by clicking the Edit button. When modifications to the node have been made, new software is generated. The node gets this updated software when it asks for updates next time; there is no method to force the update process without physical access (a node can be rebooted by turning off and on or pushing reboot button, which triggers the software update process). Node settings can be examined by clicking the Info button. From this view, the current version of the node software can also be downloaded. If a node was just created, the initial version of the software appears here. The user needs to upload this initial software to the sensor node in order to add it to OSEMA.

![Browse sensors](image)

**Figure 4.8.** The user interface of OSEMA allows managing sensors (Publication III).

Sensor nodes send data to a data server either continuously or in batches. Measurement data can be used to track the operation of the machine, for example, the movement of an overhead crane. The sample rates that can be achieved with HTTP are relatively low – in order of 10 Hz at maximum – since the processing power and network bandwidth of the nodes are limited. The maximum sample rate is a rough estimate since the sample rate was examined by doubling the rate until the sensor node crashed. The size and number of variables also affect the maximum sample rate significantly. Nevertheless, a similar sensor node is capable of up to 1-kHz sample rates when sending raw data over UDP (User Datagram Protocol) [93], but the author preferred interoperability over sample rate. Data collection also enables the development of advanced data analysis applications. For example, the usage roughness of the overhead crane could be assessed with a machine learning algorithm using an accelerometer attached to the hook (Figure 4.9).

OSEMA was employed as part of a master-level mechatronics course laboratory exercise to get feedback from the users and gain insight into...
Figure 4.9. The usage roughness of the crane was examined with a three-axis accelerometer attached to the hook (Publication 1). Students also investigated the sway of the hook with a similar measurement setup as a part of a mechatronics course.

the ease of use. The exercise consisted of configuring the sensor node attached to the hook of the crane (Figure 4.9) and analyzing the sway of the hook with this node having a three-axis accelerometer. Data was sent to a separate server plotting the sensor readings. The students were given an oral introduction to the topic, written instructions to configure the node, and they were allowed to ask for help. The sensor node was preinstalled to the hook, and all cables between the sensor and the microcontroller were connected. The exercise was mainly conducted in pairs, but some students also worked alone. The total number of students who participated in the exercise was 49, and the number of returned feedback forms was 28. It was part of the exercise to fill the questionnaire.

The user tests indicate that configuring a sensor node is rather quick. The average time to configure the node and see data plotted was less than 28 minutes. Figure 4.10 presents the ease of use assessed by the students. The average rating was 2.11 (on a scale from 1=very easy to 5=very difficult), meaning OSEMA was considered fairly easy to use.
Results and remarks

Figure 4.10. OSEMA's ease of use assessed by the students (Publication III). OSEMA was found reasonably easy to use.
5. Discussion

This dissertation proposes a general API-based architecture for building industrial digital twins. The main component of the architecture is Data Link connecting (micro-)services that implement the features of a digital twin cooperatively. Web interfaces allowing communication with these services were also examined in this dissertation. Finally, OSEMA was developed for managing sensor nodes and enabling data collection from the physical counterpart. OSEMA can be one of the services connected to Data Link, and it was part of the proof-of-concept digital twin of an overhead crane. This chapter discusses the findings and limitations of this work and presents directions for future research.

5.1 API-based Digital Twin Architecture

The main benefits of the proposed architecture result from the distributed nature of the system following the microservices architecture. First of all, microservices allow dividing the responsibilities of implementing digital twin to different teams from different companies. For example, one company specialized in simulations can implement simulation-related microservices for the digital twin, and another specialized in data analysis can implement an anomaly detection microservice. Because services can be accessed only via HTTP APIs, there are inherently clear boundaries between the responsibilities. This also forces a clear structure for digital twins. In addition, the microservices architecture allows companies to keep some of their confidential data inside a microservice hosted in their own facilities or in a cloud system. For example, data analysis models are not needed to be shared but can be kept inside a microservice. Clear boundaries between stakeholders along with keeping confidential data inside microservices helps to mitigate the problem with data ownership, which has been identified as a major issue with digital twins [38].

The use of existing systems was one of the main targets when ADTA was developed. If a holistic digital twin for an industrial machine needs to be
built from scratch, the process is laborious, slow, and costly. In addition, companies already have invested in systems, which can implement some functionalities of a digital twin, and replacing them is weakly motivated. Thus, the proposed architecture is especially suitable for factories having advanced IT systems that can already provide some functionalities for the digital twin. In addition, the architecture is an excellent solution for factories who have several stakeholders each having its own IT systems, which results in data that is scattered around and difficult to access.

A prerequisite for connecting an existing system to Data Link is that the system has an HTTP API. Nevertheless, if a system does not have an API, it can be added to digital twin by 1) implementing wrapper or protocol translator, as Hietala et al. [82] did for OPC UA, 2) providing a subset of functionalities from a separate HTTP server/interface like in Publication I with CAD files, or 3) describing how the information can be obtained from the system not connected to Data Link. The third option is likely to be employed with closed legacy systems that do not provide any interfaces and whose data is not static so that their data cannot be duplicated. In conclusion, connecting existing systems via Data Link and providing their functionalities as a digital twin features answers to research question 1: "How to build a digital twin in practice exploiting existing systems?"

Distributed systems and microservices architecture promote high modularity and scalability. Each feature of a digital twin, that is, microservice, can be replaced easily as long as the interface remains the same. This allows updating services or scaling digital twin up if, for example, more accurate simulations are needed. Even though the microservices architecture does not necessarily lead to a more robust system [55, 53], at least it is unlikely that the whole system collapses when a single service is down [56].

The purpose of Data Link is to enable communication between microservices by acting as an API gateway. In addition, the gateway provides a single access point to all digital twin features and data, which the author sees as one of the main benefits of the digital twin. Currently, data is often spread to multiple systems of the stakeholders, and the need to bring this data available using a digital twin has been recognized earlier [7, 47]. The centralized gateway also provides other benefits such as monitoring of the network traffic and access management, that is, it can be controlled who can access which microservices and, even more specifically, what services inside a microservice can be accessed. Nevertheless, neither of these features are implemented to the proof-of-concept digital twin, and, currently, a user can either access all services or none of the services. Data Link and its API gateway that provides access to all microservices solve research question 2: "How to make digital twin data and features available?"
5.2 Web interfaces for industrial communication

GraphQL and REST APIs were compared in industrial communication and as an additional interface for OPC UA servers. Scarce scientific literature about using Web interfaces in the industrial domain and the expected benefits of Web interfaces, such as interoperability, created a need for this research. Both REST and GraphQL interfaces can be used along with microservices to connect them to Data Link. GraphQL was considered a better option for several reasons: first of all, it is easier to use as multiple resources can be fetched with a single query, the structure is considered more intuitive, and it provides a GraphiQL Web tool that allows inspecting schema and assists in writing queries. Secondly, GraphQL performed better when multiple values are read because only a single query is needed. Finally, due to fewer queries, the bandwidth usage was significantly lower.

REST is a more established API design style than GraphQL. Thus, the support for different platforms is better, and almost all existing systems provide REST API instead of GraphQL. Caching is easier with REST as there is inherently a unique identifier, URL, for all resources. Nevertheless, the need for caching depends highly on the use case. For example, if one microservice returns the CAD model of the product, it is useful to cache the resource. On the other hand, with OPC UA and other status data, caching is not as beneficial because data becomes quickly obsolete.

Unique URL for each resource also provides another benefit: easier implementation of control to which service functionalities can be accessed via the API gateway. With GraphQL this requires accessing the request contents instead of using only the URL. The request execution times with single values are comparable with REST and GraphQL, and, with multiple values, GraphQL outperforms REST because of the fewer requests. Further analysis of the request execution times could not be conducted because of different software stacks. The dissertation comprehensively answers research question 3: "What are the differences between REST and GraphQL Web APIs in industrial communication?", and a more detailed comparison between REST and GraphQL can be found in Section 4.2.

This dissertation also examined "Which one is more suitable to create API for the OPC UA server, REST or GraphQL?" (research question 4). These APIs were compared to direct communication with an OPC UA server, which used OPC binary protocol and TCP. Measurements showed that direct communication offered better performance and lower bandwidth usage. Even though the measurements showed that OPC UA performs better than additional Web interfaces, this thesis recommends that a GraphQL interface is provided along with the OPC UA, because of the following reasons:

1. Interoperability. OPC UA server can be accessed from almost any Web-capable device using standard HTTP protocol and JSON. No
Discussion

need to use OPC UA specific client.

2. Ease of use. GraphQL provides a GraphiQL tool, which allows introspection of the OPC UA server structure and helps write queries.

3. Developer-friendliness. Developers are used to Web APIs, and they are more familiar with GraphQL than OPC UA.

4. Connection to Data Link. Data Link requires at the moment that HTTP protocol is used. Thus, OPC UA over TCP cannot be used. REST and OPC UA over HTTP could also be used to connect to Data Link.

REST also provides desirable features for industrial communication, such as scalability and a unique URL for each resource. Nevertheless, GraphQL was considered a better choice, because of its user-friendliness, better performance, and efficiency.

5.3 Data collection

OSEMA was developed to allow data collection from the physical twin. OSEMA provides a REST API that was implemented with the Django REST framework. The implementation was easy, as the framework makes the objects in the database accessible via API. Modifications to database objects via the API or using Web user interface triggers the software update process described in Section 3.5.3. OSEMA was added to Ilmatar overhead crane digital twin using the REST API. OSEMA allows changing measurements settings and configuring sensor nodes, and it is one of the provided digital twin features. OSEMA is already quite a large application, so it probably cannot be considered as a single microservice. Nevertheless, even though ADTA prefers the use of microservices, also systems that are monolithic can be connected to the digital twin to offer functionalities. Examples of such systems are, for example, MES and complex simulation software.

OSEMA generates software for new sensor nodes, which makes adding new sensors to machines easy. In addition, there are several configurable settings for data collection and transmission (for a comprehensive list, see Table 3.1). In conclusion, OSEMA allows data collection from machines and answers to research question 5: "How to collect data from an industrial machine for its digital twin?". In addition, as sensors can remotely be configured over the Internet using a novel software generation method (see Section 3.5.2) and secure software update process, OSEMA also covers research question 6: "How to manage data collection and change sensors’ settings over the Internet?"
5.4 Limitations

The distributed nature of API-based digital twins also invokes some limitations. First of all, performance is lower compared to systems without remote function calls over a network [53]. In addition, centralized Data Link further increases latency compared to direct communication, as shown by measurements in Section 4.1.3. The latency is still rather reasonable and suitable for sending commands to physical devices, such as "drive crane to position (x, y)", or data collection. However, the use of HTTP protocol that runs on top of TCP and communication over the public Internet makes the latency unpredictable. Thus, API gateway cannot be used with applications requiring real-time communication, such as internal control of high-speed devices. These applications should use other communication methods to implement their functionalities. For example, a machine control application could communicate directly with the machine PLC system, and the application itself could be accessed via the API gateway.

Digital twin data is distributed to several systems, which may cause consistency issues. Therefore, a master data service or database might prove useful. This problem can also be mitigated by microservices storing only the necessary data for their specific functionality. All other data is requested from other services, which prevents the use of duplicate or legacy data. For example, a simulation service stores the model and simulation parameters, but requests measurement data from another service each time the simulation is run.

Even though Data Link provides means to connect services, the technical challenge on how to build these services remains. The major problem is that technologies to implement digital twin features, such as data processing and simulation, have not yet been established [8]. Thus, more research is required, and significant effort is needed to develop digital twin features. Nevertheless, the benefit of microservices architecture is that the technologies to build services can be freely chosen [55]. Other obstacles to connecting services are the manual work required and poor documentation of the services, which, at least at the time when microservices was introduced, was based on informal documents [52]. It is recommended that the APIs of the services connected to Data Link are described in a standardized way using, for example, OpenAPI Specification [94].

One of the limitations of this thesis is that the ontology of the digital twin was not addressed in detail. The proposed DT document describes only the metadata of a digital twin, not the data structure. Even though standardized ontologies and data formats were left out of the scope of this thesis, the author recommends that systems linked via the API gateway use existing standards. The standardization of digital twin ontology is desirable, but the process should be initiated by large organizations. The standardization efforts of digital twin ontology will be followed carefully,
and applying it to ADTA is considered as future work. Nevertheless, a standardized ontology is not needed for building digital twins using the proposed API-based Digital Twin Architecture.

The request execution times of Web interfaces for an OPC UA server were measured only 50 times per case. The low number of measurements was due to the manual work required to restart the server after each measurement when the server was needed to be disconnected from the client. The standard deviation was also considered rather large, the reason of which was unclear, but was expected to be caused by the communication over a network and the internal operation of the REST interface. Nevertheless, measurements show with a very high confidence level that OPC UA is fastest in each case, and GraphQL is faster than REST when reading multiple values. The results were as hypothesized, except that using OPC UA is faster than Web user interfaces that are already connected to the OPC UA server, even though OPC UA client needs first connect to the server. This was hypothesized because OPC UA requires several handshakes before the connection is established, and one of the initial motivations to make OPC UA RESTful was to eliminate the need for these handshakes [95].

5.5 Future work

The standardization of the DT document and distribution of these documents needs to be considered as a part of future work. The current draft of the DT document is published on GitHub (https://github.com/AaltoIIC/dt-document) to initiate its standardization. There are also other standardization initiatives on describing digital twin, such as Microsoft DTDL [59] to which the DT document could be merged in the future. This dissertation does not aim to create yet another standard, but the main purpose of the DT document is to gather requirements for describing digital twins. In the future, DT document could also provide means for new services to connect to Data Link automatically and information for other services on how to access these newly added services. Finally, DT document could form a web of digital twins, in which each digital twin is described with this document. Digital Twin Web allows the discovery of digital twins and accessing their features using DT documents. The distribution of documents with Twinbase was recently introduced by Autiosalo et al. [61].

Because the centralized communication via Data Link increases latency, allowing direct communication between services that are data-intensive or require low latency should be considered in future work. In this model, Data Link would distribute credentials for the services implementing digital twin features. The performance of the API gateway could also be improved with caching (that is not yet implemented) and by streaming data immediately back to the client before the whole response has been
Discussion

Future work also includes implementing more digital twin features. The development of the features should be an iterative process based on need: With microservices, only the necessary functionalities can be added to the digital twin. In addition, the existing software should be utilized as much as possible with only slight modifications or writing an API on top of them. Large and complex systems, such as predictions or accurate simulations, could also be divided into smaller services, for example, by dividing data preprocessing, simulation, and the analysis of the simulation result to separate microservices. This would improve the modularity of the system, and promote developing multi-purpose services.

The proposed digital twin architecture should be verified within practical implementation in the industrial environment. This would help to direct the further development of the architecture and DT document. Future research could also examine the applicability of the proposed architecture outside the industry and to non-physical objects, such as organizations.
6. Conclusion

This dissertation proposed a modular digital twin architecture following microservices architecture and presented the practical implementation of the architecture with an API gateway. The architecture is based on the idea that a digital twin can be divided into separate features, initially presented by Autiosalo et al. [6]. These features can be implemented by individual Web services connected via Data Link. Each service provides an API, and Data Link routes messages to the APIs. This proposed architecture is called API-based Digital Twin Architecture (ADTA).

ADTA provides a modular way to build digital twins since new services can be added or removed based on need. ADTA allows the scalability of digital twin by replicating or replacing single services, and it is suitable for any industrial device from large industrial machines, such as overhead cranes, to simple sensors. ADTA promotes the use of existing systems, which can be connected to Data Link if they have APIs. Exploiting existing systems that already implement some digital twin features is crucial to keep costs of building digital twins at a minimum, which is a prerequisite for their wide adoption.

This dissertation also compared established REST and emerging GraphQL Web interfaces in industrial communication. These interfaces can be used to connect services implementing digital twin features to Data Link. The measurements showed that GraphQL outperforms REST in terms of request execution times and bandwidth usage with large requests. In addition, GraphQL was considered easier to use, which was mainly due to the GraphiQL tool. However, REST allows easier caching because the resources have unique URLs. This thesis recommends GraphQL over REST in industrial communication because its benefits outweigh the more difficult caching. In addition, it is suggested that GraphQL API is provided as an additional interface to access an OPC UA server allowing ease of use and compatibility with standard Web technologies.

The origins of the digital twin are accurate mirroring of the physical entity. Mirroring is still considered one of the key features of a digital twin that allows monitoring the physical product and simulations using
the real data from the field. This dissertation implemented Open Sensor Manager for keeping digital twin sync with its counterpart. OSEMA is a tool for quick deployment and configuration of sensor nodes. In addition, it enables remote management and software updates of the nodes. OSEMA can also be used to collect data for advanced data analysis applications, and the main goal of OSEMA was to allow easy data collection from the physical entity. OSEMA is available as open-source software from GitHub (https://github.com/AaltoIIC/OSEMA).

This thesis introduced a modular digital twin architecture that makes data available from a single interface and employs existing systems. In addition, this thesis compared Web technologies in industrial communication and developed a platform for collecting data from physical entities. All these contributions help to pave the way towards the Fourth industrial revolution. The fourth revolution aims to mass customization, production efficiency, and low time-to-market. These are desirable goals also from the consumer perspective: consumers will get goods suiting better to their needs, cheaper and faster. The future work includes standardization of DT document, implementing digital twin features as microservices, and validating the architecture with different digital twins and use cases in the industry.


Bibliography


API-based Digital Twins

an Architecture for Building Modular Digital Twins Following Microservices Architectural Style

Riku Ala-Laurinaho