

Master's programme in Automation and Electrical Engineering

Using OPC Unified Architecture Information Models In PubSub Over MQTT Communication Model

Iivo Yrjölä

© 2024

This work is licensed under a [Creative Commons](https://creativecommons.org/licenses/by-nc-sa/4.0/) “Attribution-NonCommercial-ShareAlike 4.0 International” license.



Author Iivo Yrjölä

Title Using OPC Unified Architecture Information Models In PubSub Over MQTT Communication Model

Degree programme Automation and Electrical Engineering

Major Control, Robotics and Autonomous Systems

Supervisor Prof. Valeriy Vyatkin

Advisors M.Sc. Jouni Aro, D.Sc. Ilkka Seilonen

Collaborative partner Prosys OPC Ltd

Date 29 April 2024

Number of pages 76

Language English

Abstract

The thesis explores the evolving landscape of OPC UA, particularly focusing on the integration of OPC UA (Open Platform Communications Unified Architecture) with MQTT in the field of industrial automation, focusing on the Publisher-Subscriber (PubSub) model. It acknowledges the challenges posed by the extensive OPC UA standard. The importance of addressing user needs and simplifying implementations is highlighted, along with the need for clearer guidelines for the implementation of different features such as PubSub.

Despite the complexity, the OPC UA specification offers robust information modeling capabilities, which can be effectively leveraged with tools like Prosys OPC UA Forge. Utilizing Prosys OPC UA Forge, the study configures an OPC UA publisher to assess the practicality and effectiveness of information models in MQTT environments.

While the PubSub specification for MQTT shows promise with recent updates providing clearer instructions and examples for topic structure, further development is needed to fully integrate extensive information modeling into MQTT messaging. The work underscores the significance of real-world examples to advance this integration, aiming to bridge the gap between operational technology (OT) and information technology (IT) systems, thereby contributing to the digital transformation in industrial automation.

Keywords OPC UA, Information models, PubSub, MQTT

Tekijä Iivo Yrjölä

Työn nimi OPC Unified Architecture informaatiomallien käyttö PubSub:in MQTT kommunikaatiossa

Koulutusohjelma Master’s Programme in Automation and Electrical Engineering

Pääaine Control, Robotics and Autonomous Systems

Työn valvoja Prof. Valeriy Vyatkin

Työn ohjaajat DI Jouni Aro, TkT Ilkka Seilonen

Yhteistyötaho Prosys OPC Ltd

Päivämäärä 29.4.2024

Sivumäärä 76

Kieli englanti

Tiivistelmä

Tämä työ tutkii OPC UA -standardin kehitystä, keskittyen erityisesti PubSub -toiminnallisuuteen OPC UA -spesifikaatiossa. Työssä tuodaan esille haasteita, joita aiheuttaa laajasta spesifikaatiosta. Työssä korostuu käyttäjien tarpeiden huomioimisen tärkeys sekä selkeämpien esimerkkien tarve OPC UA:n eri ominaisuuksien käytöstä, kuten PubSub:ista.

Vaikka OPC UA kärsii laajan spesifikaation takia vaikeasta käyttöönotosta, tarjoaa se monipuoliset mahdollisuudet tiedonmallintamiseen, joita voidaan hyödyntää tehokkaasti työkaluilla kuten Prosys OPC UA Forge. Forge helpottaa PubSub -viestinnän toteuttamisessa ja osoittaa käyttäjäystävällisen ja modernin käyttöliittymän tärkeyden OPC UA:n käytössä.

Uusimpien päivitysten myötä PubSub -spesifikaation MQTT viestintä vaikuttaa entistä lupaavammalta se kuitenkin tarvitsee edelleen kehitystä, jotta OPC UA:n tietomallit voitaisiin integroida MQTT-viestintään. Työ korostaa käytännönesimerkkien merkitystä, jotta PubSub:ia voitaisiin hyödyntää, samalla pyrkien pienentämään operatiivisen teknologian (OT) ja informaatioteknologian (IT) välistä kuilua. Samalla edistetään automaatioteollisuuden digitaalisaatiota.

Avainsanat OPC UA, informaatiomallit, PubSub, MQTT

Preface

I want to thank both of my advisors for their guidance and feedback throughout the thesis process. Additionally, I want to express my sincere appreciation to Prosys OPC for their support and assistance in completing this work.

I also want to express my gratitude to my family and friends for their continuous support and belief in me throughout this work.

Matinkylä, 29 April 2024

Iivo J. Yrjölä

Contents

Abstract	ii
Abstract (in Finnish)	iii
Preface	iv
Contents	v
Abbreviations	x
OPC UA Terminology	xi
1 Introduction	1
1.1 Background and motivation	1
1.2 Objectives	2
1.3 Research Methods	2
1.4 Structure	2
2 OPC UA	4
2.1 Overview	4
2.2 Address Space Model	5
2.2.1 NodeClass	5
2.2.2 Attributes	8
2.3 Information Models	8
2.3.1 Object model	9
2.3.2 TypeDefinitions	10
2.3.3 Event model	11
2.3.4 Roles	12
2.3.5 Interfaces and AddIns	12
2.3.6 Graphical representation	13
2.3.7 Cloud Library	14
2.4 Services	15
2.5 Security	16
2.6 Communication	16
2.6.1 Client-Server	16
2.6.2 PubSub	17
3 MQTT	20
3.1 Overview	20
3.2 MQTT Essentials	20
3.2.1 MQTT topic structure	21
3.2.2 MQTT features	22
3.2.3 MQTT 5 features	22

4	OPC UA PubSub	24
4.1	Overview	24
4.2	PubSub information model	27
4.3	Topic structure in OPC UA PubSub	28
4.4	DataSetMessage	29
4.4.1	VariableDataSet	29
4.4.2	EventDataSet	30
4.5	Discovery messages	30
4.5.1	DataSetMetaData	31
4.6	Message header content	31
4.6.1	Header layouts	34
5	Experimental PubSub setup	35
5.1	Target of the PubSub configuration	35
5.2	Architecture of the setup	35
5.2.1	UaModeler	37
5.2.2	Prosys OPC UA Simulation Server	37
5.2.3	Prosys OPC UA Forge	37
5.2.4	Mosquitto Broker	38
5.2.5	Prosys OPC UA Browser	38
5.3	Information model	38
5.4	Simulated variables	41
5.5	Configuring OPC UA Forge data	41
5.5.1	Connecting Forge to Simulation Server	42
5.5.2	Address Space configuration	42
5.6	Mosquitto broker	46
5.7	Configuring Publishers to Forge	46
5.7.1	VariableDataSet	47
5.7.2	EventDataSet	48
5.7.3	Configure a connection to MQTT broker	49
5.7.4	Configure WriterGroups	50
5.7.5	Configure DataSetWriters	50
5.8	Subscribing to the topics	51
6	Results	53
6.1	Topic structure	53
6.1.1	Default OPC UA topic structure	53
6.1.2	UNS topic structure	54
6.2	Message structure	55
6.2.1	MetaData messages	55
6.2.2	VariableDataSet messages	56
6.2.3	EventDataSet messages	57

7 Discussion	59
7.1 Topic role in MQTT messaging	59
7.2 Message contents	60
7.3 Forge’s capabilities	61
7.4 Future directions	61
8 Conclusion	63
References	64
Appendix A	66
JSON Data Message description	66
Appendix B	68
JSON MetaData Message description	68
Appendix C	72
Full MQTT Data and MetaData messages generated by Forge	72

List of Figures

1	The structure between basic ReferenceTypes in the OPC UA Address Space. [2].	7
2	Instance created from TypeDefinition.	7
3	Information Models architecture [17].	9
4	OPC UA object model.	9
5	Inheritance and Extension in OPC UA TypeDefinitions: Building Blocks for Information Modeling.	11
6	Graphical representation of OPC UA Nodes.	14
7	Client-server architecture.	17
8	Broker-based PubSub	18
9	Broker-less PubSub	19
10	PubSub in automation industry architecture.	25
11	Overview of the architectural change in industrial communication.	26
12	PubSub NetworkMessage structure and key configuration parameters.	28
13	System architecture of the experimental setup.	36
14	ObjectTypes of the created Information model.	39
15	VariableTypes of the created Information model.	40
16	ReferenceType of the created Information model.	41
17	Diagram of the Address Space created.	43
18	Configuring a new Node into the Address Space.	44
19	Instantiated type in the Address Space.	45
20	Mapping of attributes in Forge.	46
21	Re-usable event template.	48
22	Event Generator configuration.	49
23	PubSub over MQTT connection configurations.	50
24	PubSub variable messages received by OPC UA Browser.	57
25	PubSub event messages received by OPC UA Browser.	58

List of Tables

1	Overview of OPC UA NodeClasses.	6
2	Base NodeClass Attributes (BaseAttributes).	8
3	The default topic levels defined in PubSub specification. [11]	29
4	Simulated Variables Overview	41
5	DataSets for UNS topic structure.	48

Abbreviations

A&C	Alarms and Conditions
CESMII	Clean Energy and Smart Manufacturing Innovation Institute
CPPS	Cyber-physical production systems
DA	Data Access
ERP	Enterprise Resource Planning
HA	Historical Access
IIoT	Industrial Internet of Things
IT	Information Technologies
LWT	Last Will and Testament
M2M	Machine-to-Machine
MES	Manufacturing Execution System
MOM	Message Oriented Middleware
MQTT	MQTT is no longer considered an acronym. (Previously, Message Queuing Telemetry Transport or MQ Telemetry Transport)
OASIS	Organization for the Advancement of Structured Information Standards
OEE	Overall Equipment Effectiveness
OPC UA	(Open Platform Communication) OPC Unified Architecture
OT	Operation Technologies
QoS	Quality of Service
PubSub	Publish-Subscribe
SCADA	Supervisory Control and Data Acquisition
SKS	Security Key Service
SOA	Service-oriented Architecture
UNS	Unified Namespace

OPC UA Terminology

Address Space	Collection of information that a Server makes visible to its Clients. [1]
Attribute	Primitive characteristic of a Node. [1]
Information Model	OPC UA representations of model and types for different use cases.
Node	Fundamental component of an Address Space. [1]
NodeId	Nodes are unambiguously identified using a constructed identifier called the NodeId. [2]
NodeClass	Class of a Node in an Address Space. NodeClasses define the metadata for the components of the OPC UA object model. [1]
Object Model	The OPC UA Object Model has been designed to provide a standard way for Servers to represent Objects to Clients. [2]
PubSub	OPC UA specification to describe publish-subscribe communication architecture

1 Introduction

1.1 Background and motivation

In the rapidly evolving field of industrial automation, the development of new communication protocols forms the backbone of modern manufacturing and control systems. Among these, OPC UA (Open Platform Communications Unified Architecture) stands out as the most promising and widely adopted standard, enabling seamless interoperability and data exchange across diverse platforms and devices [3] [4].

The integration of communication technologies plays an important role in enhancing system efficiency and interoperability. The goal is to achieve seamless integration between operational technology (OT) and information technology (IT) systems. Previously OPC UA has only utilized client-server communication restricting its usage. The communication was done through coupled connections between IT-OT. IT refers to the use of computer systems, software, and networks to manage and process information. IT plays a crucial role in the automation sector by providing the necessary applications and infrastructure for various automated processes. It takes care of storing and processing the data and delivering the new information. OT refers to the hardware and software technologies used to monitor and control physical processes, devices, and infrastructure. While IT focuses on managing and processing data through computer systems and networks, OT is specifically concerned with the technologies that directly interact with the physical world in industrial settings.

OPC UA has acknowledged the need for a more interconnected communication architecture. The integration of OPC UA with MQTT (no longer an acronym in this context) in the Publisher-Subscriber (PubSub) model represents a significant leap in addressing the requirement. MQTT is a lightweight and scalable communication architecture that is widely adopted across smart devices. This integration promises to unlock new levels of performance and flexibility to meet the evolving demands of Industry 4.0 where all devices, sensors, and products are more interconnected and smart [4].

The industry needs a more interconnected architecture to be more re-configurable, driven by market demands for increasingly customized products with shorter delivery times [5] [6]. Another use case is cloud computing and services, which require access to raw process values to enable machine learning for process optimization and predictive maintenance reporting for manufacturers [5] [7]. Cloud communication requires secured and optimized messaging to avoid data snooping and overloading the communication network when a large amount of data is sent to the cloud [6]. This concept, known as the Industrial Internet of Things (IIoT), focuses on bridging the gap between IT and OT [5]. These advancements underscore the importance of enhanced metadata from the process, signifying digitalization or digital transformation and highlighting the integration between the physical and informational worlds [6].

However, the journey toward achieving a fully integrated OPC UA and MQTT ecosystem has its complexities. The extensive OPC UA standard presents a challenge for adaptation, requiring a deep dive into its comprehensive specifications to unlock its full potential. This thesis is motivated by the need to clarify these complexities, offer

example implementation, and provide clear explanations for leveraging the PubSub model over MQTT communication within the field of industrial automation.

By addressing these challenges, this thesis aims to explore the practicalities and advantages of using OPC UA's information models within MQTT environments. An experimental setup was created with modern integration software, OPC UA Forge, to test OPC UA PubSub messaging. Thus, contributing to the digital transformation journey of industrial automation. The goal is to bridge the gap between OT and IT, creating a unified ecosystem where data flows freely and securely, and driving progress towards a smarter, more connected industrial future.

1.2 Objectives

The main objective of this thesis is to explore the integration of OPC UA Information Models with OPC UA PubSub over MQTT communication within the industrial automation field. This study aims to demonstrate the complexities of this integration. The specific objectives include:

- To conduct a detailed examination of the OPC UA specifications, particularly the PubSub over MQTT model, to understand its capabilities and limitations in the context of OPC UA.
- Analyzing the significance of information models within both OPC UA and MQTT environments, assessing their efficacy in facilitating data representation and exchange across diverse systems to determine their suitability for various applications.
- Evaluating the Forge application's capabilities as a configuration tool for OPC UA publishers, demonstrating its features, functionalities, and user-friendliness in facilitating the setup of OPC UA publishers for efficient communication purposes.

1.3 Research Methods

An extensive review of existing literature on OPC UA standards, MQTT protocols, and their applications in industrial automation.

A practical setup was created using Prosys OPC UA Forge to configure an OPC UA publisher and assess the practicality of information models in PubSub over MQTT communication. This experimental setup aims to provide hands-on experience with OPC UA PubSub functionalities. By integrating theoretical knowledge with practical experimentation, this research methodology aims to offer a well-rounded exploration of OPC UA PubSub communication.

1.4 Structure

This master's thesis systematically explores the integration of OPC UA and MQTT protocols, focusing on OPC UA PubSub. The introduction outlines the study's

motivation, objectives, research methods, and structure. Chapters 2 through 4 delve into OPC UA, MQTT and OPC UA PubSub fundamentals, covering its architecture, information models, and messaging structures.

Chapter 5 introduces the experimental setup for PubSub configuration, detailing the architecture and components involved. This leads to Chapter 6, where the configuration process is outlined, including information model setup. Chapter 7 presents the results and discusses findings, focusing on message and topic structures within the PubSub communication and suggests future research directions. Finally, the Conclusion chapter summarizes key findings.

2 OPC UA

This section will cover the basics of OPC UA, describing the key elements of the OPC UA specifications. The core concepts of OPC UA are covered in OPC UA specification's parts 1. to 5. and part 14.[1][8][2][9][10][11]. Additionally, these specifications can be accessed from online [12].

2.1 Overview

OPC UA (Open Platform Communications Unified Architecture) stands as a leading interoperability standard, consisting of multiple specifications covering different topics of the architecture. Parts 1 through 5 cover the core topics of OPC UA. OPC UA enables flexible, secure, and reliable data exchange in industrial automation and beyond. Introduced in 2006 (released in 2008) to improve upon the earlier OPC Classic solution, OPC UA merged all specifications into a single and comprehensive standard [13]. As mentioned before, OPC UA is the dominant standard for meeting the requirements of Industry 4.0, which encompasses current trends in the automation industry such as machine-to-machine communication, security, the Industrial Internet of Things (IIoT), Cyber-physical production systems (CPPS) and information transparency. As a result of its successful development process, OPC UA can be used in all industrial fields, including industrial sensors, actuators, control systems, SCADA, MES, and ERP, thereby reducing the effort required for data exchange between these systems [1].

Unlike its predecessor, OPC Classic, OPC UA is platform-independent. Additionally, it extends data modeling capabilities through the use of meta-information and well-designed information models [1]. OPC UA supports multiple communication protocols and includes various security features for secure web communication.

OPC UA is an open standard developed and maintained by the OPC Foundation. The standard has been developed in collaboration with industry vendors, end-users, and software developers [13]. Hence, the standard is practical and responsive to the needs of its users. The OPC Foundation is responsible for ensuring compliance with new specifications through certification testing.

OPC UA consists of three basic data models that provide comprehensive support for multiple use cases. First, there is Data Access (DA), which is used to read, write and subscribe to data [14]. This model is mainly used to communicate between PLCs and SCADA. The read and write methods serve as basic communication between client and server, while subscriptions allow Clients to be notified of all changes in the subscribed data. The next data model is Historical access (HA), which provides access to read and update historical measurement values [15]. This enables applications to generate trend data of the process history. Finally, there are Alarms and Conditions (A&C) [16], which enable the push and sending of notifications and alarms. These are useful for generating reports of the manufacturing process and for notifying of changes. Methods can also be considered as one data model too, enabling a service-oriented architecture that makes OPC UA extensible and customizable for different needs.

2.2 Address Space Model

Address Space is a key feature of OPC UA, describing how data is structured within an OPC UA server. A uniformly represented Address Space guarantees that every OPC UA client can access the data in servers and interact with them. It defines the basic rules for structuring and handling data.

OPC UA Address Space uses object-oriented techniques such as type hierarchies and inheritance [1]. Thus, clients can handle instances of the same type uniformly, allowing the development of client applications without requiring knowledge of specific information. Although the Address Space is defined on the server side, clients can access a small piece of the server's information without needing to understand the entire Address Space model, thanks to their knowledge of the specifications of the Address Space structure. The Address Space forms a fully meshed network, enabling information to be linked in various ways. With these capabilities, OPC UA becomes a very powerful tool for modeling and exposing complex data and systems.

OPC UA is designed to be extensible and thus it is under continuous development with regular updates and even new specifications are presented. Hence, the Address Space receives new features and rules like any other software. The OPC Foundation ensures that these specifications remain compatible with previous versions and do not conflict with existing ones. To stay updated with the specifications, one must follow resources such as the OPC Foundation homepage [13]. Nonetheless, understanding the Address Space does not necessarily require knowledge of all the specifications.

2.2.1 NodeClass

The Address Space is structured with Nodes [2]. A Node represents a point of data within the Address Space and is an instance of a NodeClasses, which varies depending on the purpose of the instantiated Node. There are eight NodeClasses which are listed in the following Table 1.

Table 1: Overview of OPC UA NodeClasses.

Name	Description
Object	A Node belonging to the Object NodeClass represents an instance of a real device, process, or software.
Variable	A Variable Node represents an instance of a value that can be written, such as a set-value, and a value that can be read like a measurement value.
Method	A Method Node is a callable function that executes actions, such as starting or stopping a process
ObjectType	Every Object Node has a reference to a TypeDefinition, which is a Node that is part of ObjectType NodeClass.
VariableType	Every Variable Node has a reference to a TypeDefinition, which is a Node that is part of VariableType NodeClass.
ReferenceType	Nodes are connected with references, which describes the relationships between them. Each reference is associated with a ReferenceType that adds semantic meaning to the relationship. References can be hierarchical or non-hierarchical. Hierarchical references mean a parent-child relationship, while non-hierarchical references create arbitrary associations.
DataType	DataType Nodes are used to define the type of the actual value, such as string, float, Boolean, etc.
View	A View is a subset of the Address Space, showing only nodes relevant to the Client. This allows the Address Space to be focused on specific users with specified views.

Basic ReferenceTypes can be categorized as hierarchical or non-hierarchical, as illustrated in Figure 1 below.

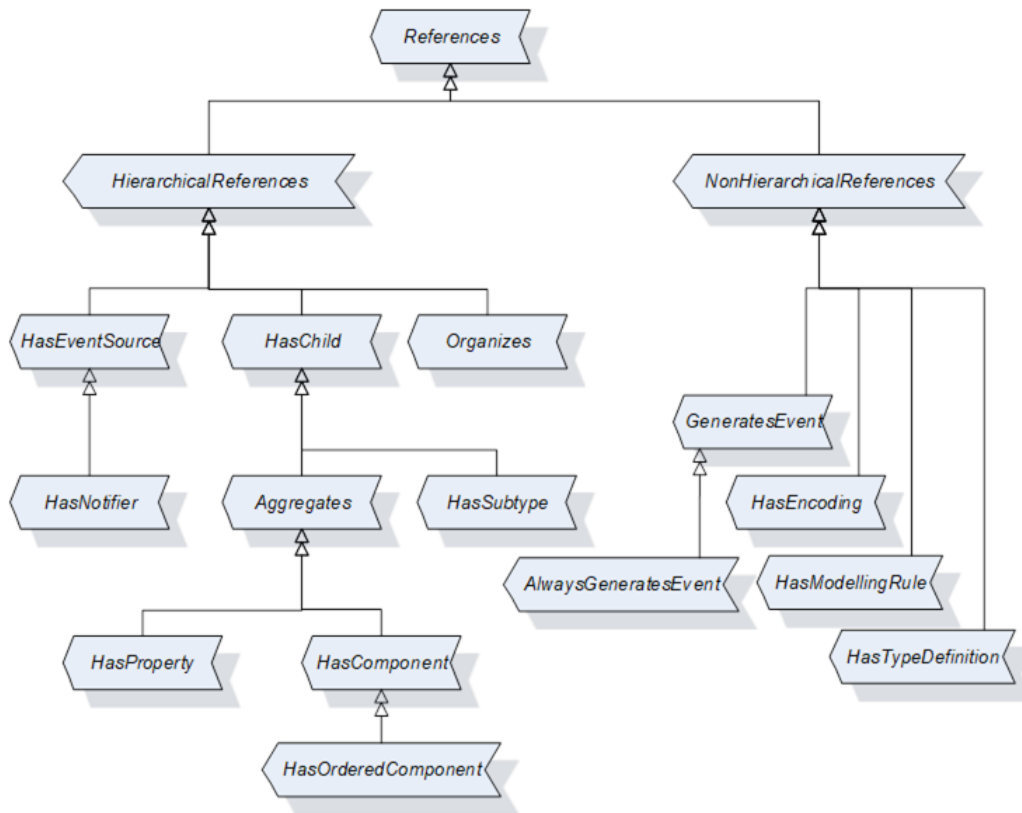


Figure 1: The structure between basic ReferenceTypes in the OPC UA Address Space. [2].

A brief example illustrates the usage of various NodeClasses within an instance. As depicted in the following Figure 2, the ObjectTypes form the structure, alongside linked variables and methods. An instance of the PetType, named My_Dog, is created. The My_Dog object adheres to the structure of the PetType but also extends the type with an additional variable.

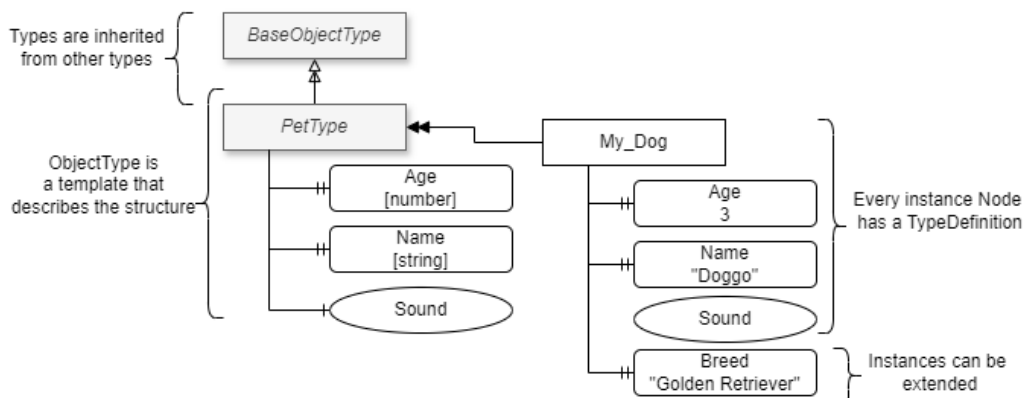


Figure 2: Instance created from TypeDefinition.

2.2.2 Attributes

Each of these NodeClasses has a fixed set of Attributes depending on the class [17]. In simpler terms, Attributes describe the Node, with some being mandatory to provide information, while others are optional. The Attribute field can be left empty when defining a new instance of the Node. The Most important Attribute is the NodeId, which is mandatory for every Node. This Attribute serves to identify the Node within the Address Space, and it is unique for every Node. A comprehensive table of basic Nodes is provided in Table 2 below.

Table 2: Base NodeClass Attributes (BaseAttributes).

Attribute name	Use	Data Type
NodeId	Mandatory	NodeId
NodeClass	Mandatory	NodeClass
BrowseName	Mandatory	QualifiedName
DisplayName	Mandatory	LocalizedText
Description	Optional	LocalizedText
WriteMask	Optional	AttributeWriteMask
UserWriteMask	Optional	AttributeWriteMask
RolePermissions	Optional	RolePermissionType[]
UserRolePermissions	Optional	RolePermissionType[]
AccessRestrictions	Optional	AccessRestrictionsType

2.3 Information Models

To fully leverage the benefits of information modeling, the OPC Foundation started to create specifications aimed at describing the structure of assets. These specifications are known as Information Models [10]. The objective is to establish standardized information models for devices, systems, and other applications across various industries. This approach ensures that end-users' models are not limited to a project or vendor-specific configurations but are instead reusable and universally recognized Information Models with consistent basic structures. Naturally, vendors can extend OPC UA Information Models with their own vendor-specific fields. Consequently, devices and systems of the same type should adhere to similar type structures, with only vendor-specific information added. The information model architecture of OPC UA is illustrated in Figure 3 below.

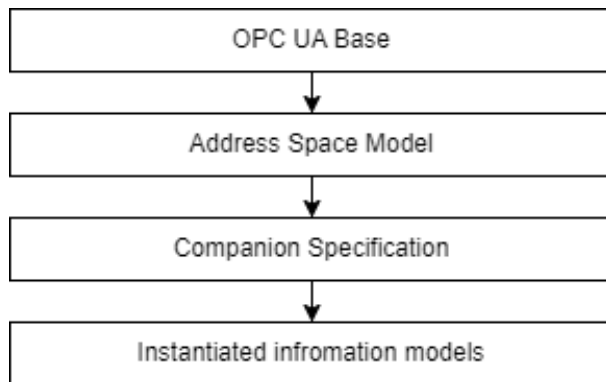


Figure 3: Information Models architecture [17].

Semantic and enriched data allow OPC UA clients to execute complex tasks through logical use of the provided semantics and data [17]. This encompasses automated integration and processing of data, as well as smart communication between machines. Clients can have specific information about the type of device and its location, among other details.

2.3.1 Object model

The fundamental concept is the Object Model, which defines Objects with Variables and Methods [2]. Relationships to other Objects are expressed through references, as illustrated in Figure 4 below.

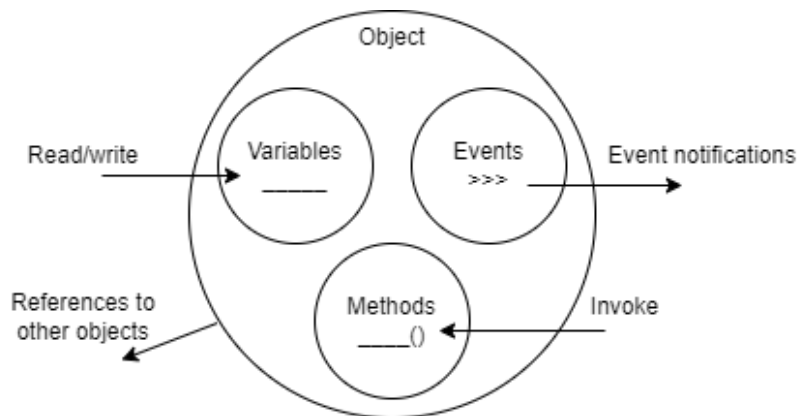


Figure 4: OPC UA object model.

As described in the Address Space Chapter 2.2, the Object Node creates the structure and represents a system, device, or similar entity. The Variable Nodes represent values. Two types of Variables are defined: Properties and DataVariables.

Properties are defined by the server, describing the characteristics of Objects, DataVariables, and other Nodes [2]. While Properties may seem similar to Attributes, they have a key difference. Attributes are inherited from NodeClass and define

additional metadata, whereas properties characterize, what the Node represents. For instance, Properties could specify the Engineering Unit.

Properties are not allowed to have Properties defined for them. To easily identify Properties, the BrowseName of a Property shall be unique in the context of the Node containing the Properties (see Chapter 5.6.3 of OPC UA specification part 3. for details [2]). A Node and its Properties shall always reside in the same Server.

DataVariables' purpose is to represent the content of an Object [2]. A great example is an object that represents a person's profile. The object would contain a DataVariable with personal information like name, age, and gender. DataVariables can also be complex, meaning they can have separate components of the data structure to represent, for example, "address" and "phone number".

Methods are like "lightweight" functions and can be compared to methods of a class in object-oriented programming [17]. They are invoked by a client, triggering functions on the server, and finally returning a value to the client. Methods can have multiple input and output arguments. They are instances of Nodes classified as the Method NodeClass, which contains the metadata of the arguments and their behavior. A Method's lifecycle begins with the client invocation and ends with the return value sent back to the client.

2.3.2 TypeDefinitions

Basically, the Address Space is constructed with instances created according to TypeDefinitions. These TypeDefinitions do not contain actual data or information but serve as templates for instances representing real-world components with data. Instances are linked with the TypeDefinition using the HasTypeDefinition reference [10]. TypeDefinition is mandatory, ensuring that instances inherit the Attributes specified in the TypeDefinition. Some TypeDefinitions have the Attribute IsAbstract set to True, indicating that these Nodes are not applicable to be instantiated. Instead, they are used solely to provide structure for modeling and must have a Subtype that can be instantiated. More on Subtypes later in this chapter.

OPC UA also enables complex TypeDefinitions, meaning that the TypeDefinition Node contains references to other Nodes as part of the TypeDefinition [2][17]. There are ModellingRules that define how these Nodes should be instantiated, indicating whether the Referred Nodes must be instantiated or not. A key point in ModellingRules and information modeling is that the ModellingRules are not allowed to be loosened when Subtyping a complex TypeDefinition.

The next concept of information modeling is subtyping. This means that the Subtype refers to another TypeDefinition and inherits the structure of the supertype, allowing the Subtype to add additional content to the type [2]. The benefit is that clients can handle the Subtype with only knowledge of the supertype. The instance of the supertype can be replaced by an instance of the Subtype. Subtyping is a widely used modeling concept, and information models are mainly based on Subtyping. The following figure, Figure 5 demonstrates subtyping. A MotorType is SubTyped with type ElectricMotorType. Then two instances are created from the ElectricMotorType to showcase the inheritance of the structure.

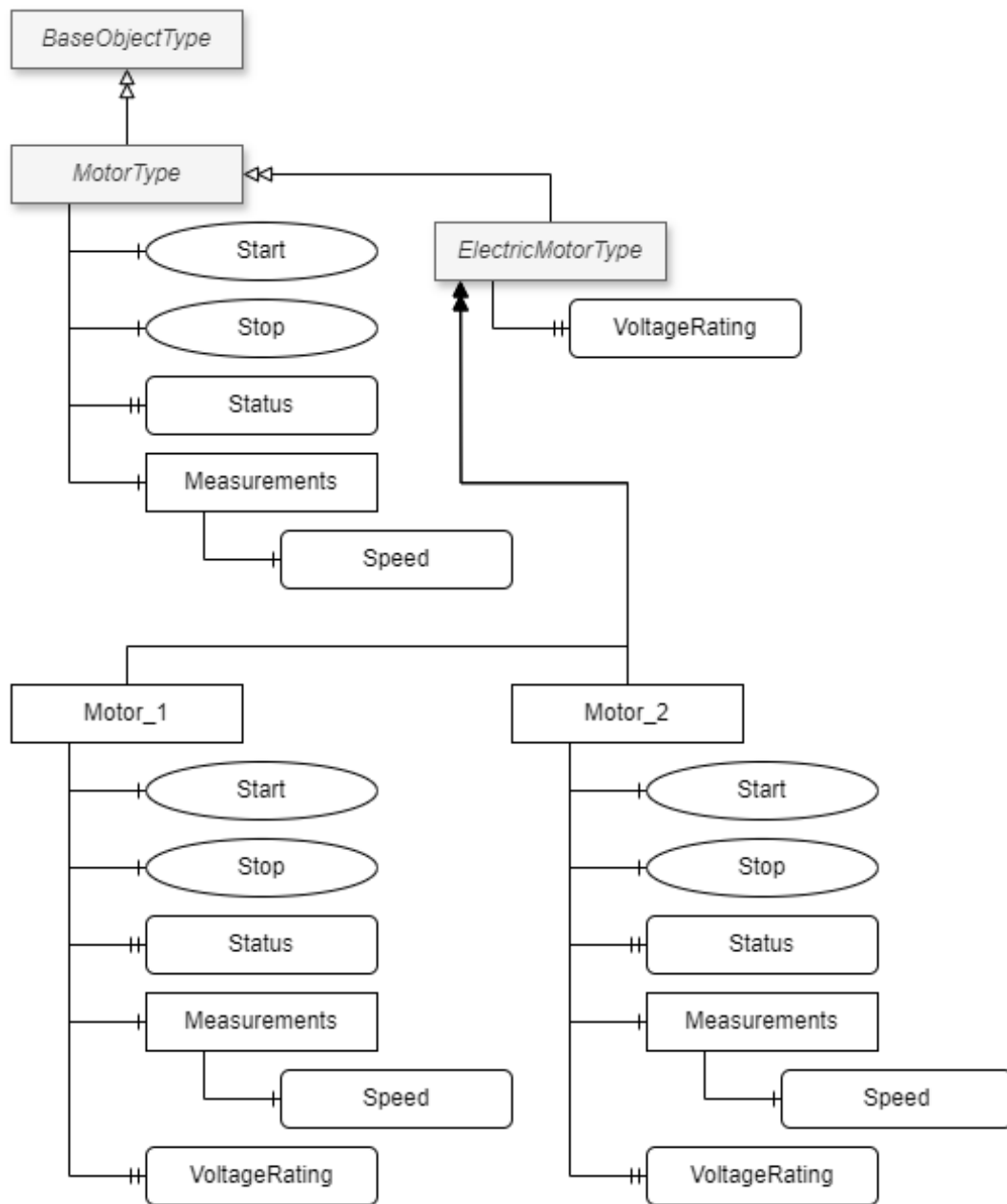


Figure 5: Inheritance and Extension in OPC UA TypeDefinitions: Building Blocks for Information Modeling.

2.3.3 Event model

In OPC UA, the event model is used to represent and notify information about events occurring in the system [2]. This model defines a standard approach to describe and report events, including notifications, alarms, and other conditions that may be of interest to users or systems.

The event model in OPC UA is based on a few key concepts. Firstly, there are event sources, which are objects or components within the system capable of generating

events. These sources can be monitored by event listeners, which are other objects or components that are interested in receiving notifications about events.

Events themselves are represented as objects with specific EventFields, such as a timestamp, severity level, and a message describing the event [2]. OPC UA also supports the concept of event types, which are templates or definitions that describe the attributes and behavior of events that may occur within the system.

In general, the event model in OPC UA provides a flexible way to represent and communicate information about events within a system, making it easier for users and systems to monitor and respond to changes and conditions.

2.3.4 Roles

Roles are used to define and manage access control to the system's resources [2]. A role is a named set of permissions that determines what actions a user or group of users is allowed to perform within the system.

Roles can be assigned to individual users or groups of users, and can be used to control access to specific resources within the system, such as data Nodes, methods, or events. For example, a role might be defined to allow read-only access to a certain set of data Nodes, or to allow a user to execute specific methods but not others.

Roles can also be used in combination with policies, which define how access control decisions are made based on a set of rules or criteria. Policies can be used to enforce different levels of access control depending on factors such as the user's identity, the type of device or application being used, or the time of day.

Overall, the use of roles in OPC UA provides an adaptable way to manage access control within the system, ensuring that users are only allowed to perform the actions that they are authorized to do. This increases the security and reliability of the system.

2.3.5 Interfaces and AddIns

The interface model and AddIn model are two related concepts of information modeling. They are used to define and extend the behavior and capabilities of the system.

The Interface model in OPC UA is a standardized approach for representing the behavior and capabilities of objects within the system, using a predefined set of structures [2]. It provides a consistent way to interact with different objects within the system, making it easier to develop and integrate different components and systems. A typical interface could be an identification interface, containing Variables for product number, model, and software version, for instance. Each interface should only be applied once to a single Node. Interfaces are subtypes of BaseInterfaceType.

The AddIn model in OPC UA is a concept designed to expand object models by adding features or feature-sets represented by ObjectTypes to existing objects [2]. An AddIn is a component designed to provide additional functionality that is not available in the standard ObjectType. AddIns do not have a specific supertype; instead, instances are identified as an AddIn through the use of the HasAddIn Reference or subtype. AddIns can introduce new features and capabilities to the system, such as motors, machines, tools, etc.

All in all, the main advantage is that Interfaces and AddIns allow standard extensions to types without subtyping. Thus, they provide a versatile way to define and extend the behavior and capabilities of the system, making it easier to develop and integrate different components and systems, and to add new features and functionality as needed, such as device health diagnostics.

2.3.6 Graphical representation

OPC UA has established specific symbols for use in diagrams that present an OPC UA information model. These notations will be used later in this work to represent the information model used in the experimental setup. The symbols are outlined in Figure 6 below:

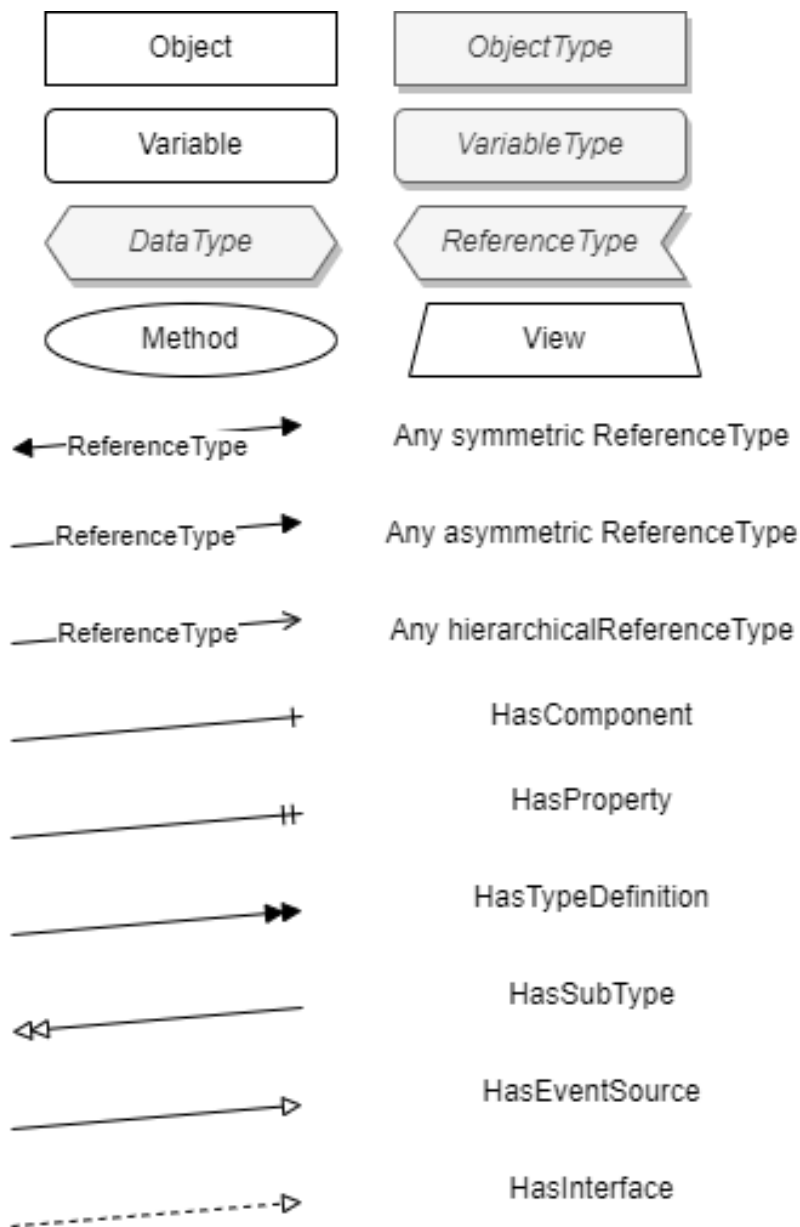


Figure 6: Graphical representation of OPC UA Nodes.

2.3.7 Cloud Library

The Clean Energy and Smart Manufacturing Innovation Institute (CESMII) and OPC Foundation working group have developed and deployed an Internet-hosted database known as the UA Cloud Library to store OPC UA information models [18]. This library is currently operational online, providing a RESTful interface for uploading, downloading, and querying OPC UA Address Spaces.

The UA Cloud Library provides a REST interface for managing OPC UA Address Spaces. Clients can utilize existing information models within the Address Space, although final configuration typically requires online access to the Server.

Additionally, the Cloud Library supports partial Address Space uploads. Stored Address Spaces are identified using unique identifiers such as ProductInstanceUri or ApplicationInstanceUri [18].

The Cloud Library enables various use cases, such as application configuration, specification compliance verification, retrofitting machines with OPC UA, downloading Address Spaces from the UA Cloud Library into blank UA server instances, and accessing existing Address Spaces for engineering support purposes [18]. Additionally, it could potentially support PubSub communication by serving as a platform for exchanging Address Spaces between publishers and subscribers.

2.4 Services

OPC UA adopts a service-oriented architecture (SOA), where functionality is organized into a set of reusable services, known as Service sets [17][9]. Each service is responsible for a specific task, such as reading or writing data, and is accessed through a clearly defined interface. In OPC UA, these services are implemented by the server, which provides access to underlying data and devices. Clients can then access these services to read or write data, subscribe to data changes, and execute other operations. It's important to note that OPC UA Services are abstract descriptions and they do not represent a specification for implementation.

The service-oriented architecture of OPC UA offers several advantages. Firstly, it allows for great interoperability, since services are accessible through standardized interfaces by multiple clients. Secondly, it makes code reusable, thereby reducing the need for redundant code and improving application development efficiency. Thirdly, it ensures scalability, enabling OPC UA to support large and complex systems. Moreover, the service-oriented architecture allows for new services to be added or existing ones to be modified. This can be done without affecting the rest of the system, making the system flexible.

The following list will cover a few Service Sets to provide examples of the services: [9].

- **Discovery Service Set:** Used to discover the Endpoints implemented by a Server and read the security configuration for those Endpoints. Discovery Services are in a dedicated Discovery Server. Shortly, the process begins with the server registering itself to the Discovery Server using RegisterServer(). Then, the client can find it with FindServers(), and finally, GetEndpoints() provides all the server's Endpoints.
- **NodeManagement Service Set:** Used to make changes to the Address Space. It is mainly used to add and delete Nodes and References to and from the server Address Space.
- **View Service Set:** Allows clients to navigate through the Address Space or through a View. The term "navigation" is often replaced with the word "browse".

- **MonitoredItem Service Set and Subscription Service Set:** Used to manage the MonitoredItems and Subscriptions. The server provides notifications regarding these MonitoredItems and Subscriptions based on attribute value changes and events.

Other Service Sets include SecureChannel, Session, Query, Attribute and Method.

2.5 Security

OPC UA security aims to provide authentication, authorization, integrity, confidentiality, availability, and auditability in its communication between clients and servers [8]. This means that OPC UA specifies strategies to ensure who can access the data, how to keep the data protected from eavesdroppers, and how the messages can be trusted on both ends. However, OPC UA does not specify the circumstances under which various security mechanisms are required because these circumstances can vary from system to system and from industry to industry. Therefore, it is up to the designers and developers of the system to determine the security requirements and mechanisms appropriate for their specific application. This includes deciding which security mechanisms to use, how to configure them, and when to apply them.

OPC UA provides a comprehensive set of security features, including encryption, message signing, user authentication, and access control. These features can be used to build a secure OPC UA system tailored to the specific security requirements of the application. It is important to design and implement a secure OPC UA system to protect against unauthorized access, tampering, and theft of data.

2.6 Communication

OPC UA is designed to address communication needs across various levels and applications within the automation industry [1]. Communication requirements differ depending on the use case, ranging from fast and lightweight needs at the field level to large data availability in cloud applications. Thus, OPC UA supports multiple compatible communication protocols to meet these diverse requirements. Previously, OPC UA relied only on the client-server communication model, which was also present in its predecessor OPC Classic. However, in 2018, OPC Foundation introduced a new extension: the publish-subscribe communication model, known as PubSub [13].

2.6.1 Client-Server

In the OPC UA client-server model, the client and server are separate entities that communicate with each other over a network. The client initiates the communication by sending requests to the server, which then responds with the requested data or performs the requested action. A single client can communicate with multiple servers, and vice versa. Additionally, the architecture can be extended with combined client-server entities, enabling bidirectional communication capabilities. This setup, known as a chained client-server architecture, enables devices to both request data as clients

and serve data as servers. The client-server model is particularly useful for scenarios involving a small number of entities and tasks such as fetching irregular reports and invoking methods to control processes.

Client-server communication involves several steps. First, the client must establish a connection with the server. Following this, the client can start a session, allowing it to browse the server's Address Space, read and write data, invoke methods, and subscribe to data. To perform these actions, the client sends requests to the server, which gives a response after needed actions to the request. Finally, once the communication tasks are completed, the client terminates the session with the server. The client-server communication model operates on a session-based synchronous communication approach. The following Figure 7 demonstrates the client-server communication and a combined client-server entity.

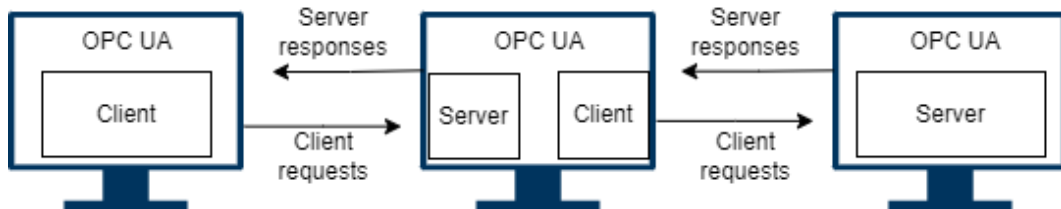


Figure 7: Client-server architecture.

Overall, the OPC UA client-server communication model provides a robust way to exchange data and method calls between devices and systems in an industrial automation and control environment. However, client-server communication may face limitations in message load optimization, speed, and scalability. To address these needs and facilitate centralized data exchange in cloud-based environments, the PubSub extension was introduced.

2.6.2 PubSub

PubSub extends OPC UA capabilities to field-level and cloud communication, offering lightweight, high-performance, and real-time message transfer [11]. It enables easy and scalable communication with the cloud by specifying the publish and subscribe communication model. The PubSub specification defines three components: publisher, subscriber, and message-oriented middleware (MOM). The publisher, often a server, serves as the data source, while the subscriber, typically a client, consumes the data. MOM is located between the publisher and subscriber, enabling the message exchange.

MOM can be defined as two different types: broker-based or broker-less [11]. In a broker-less architecture, the network infrastructure (e.g., routers, switches, and bridges) functions as MOM, making it suitable for local networks and controller-to-controller communication. In the broker-based architecture, a message broker works as MOM, managing message delivery between the publisher and subscriber. This architecture is well-suited for cloud communication. For the purposes of this thesis, focus is on the broker-based architecture.

In PubSub communication, OPC UA data is published by publishers, while subscribers consume the data by subscribing to the published data. This communication model is scalable since publishers do not need to know anything about the subscribers or their existence. PubSub communication is asynchronous and session-less, meaning that subscribers can receive data from the publisher without the need to be simultaneously connected. This feature is particularly useful for IIoT applications. The architecture can be broker-based or broker-less.

The following Figures 8 and 9 illustrate the architectures.

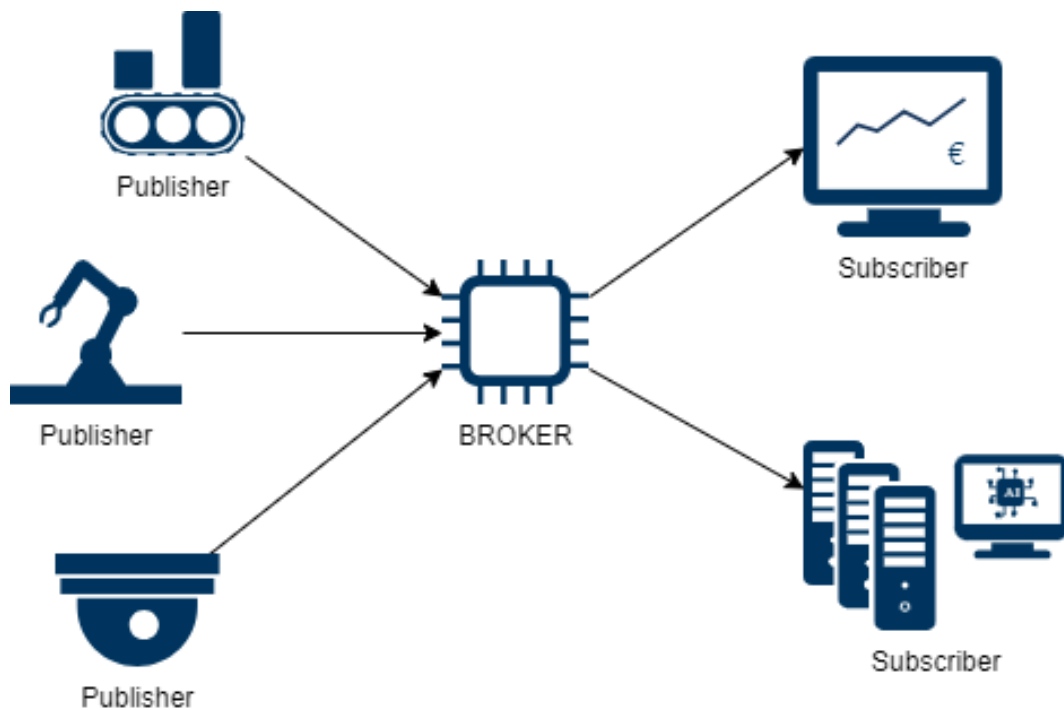


Figure 8: Broker-based PubSub

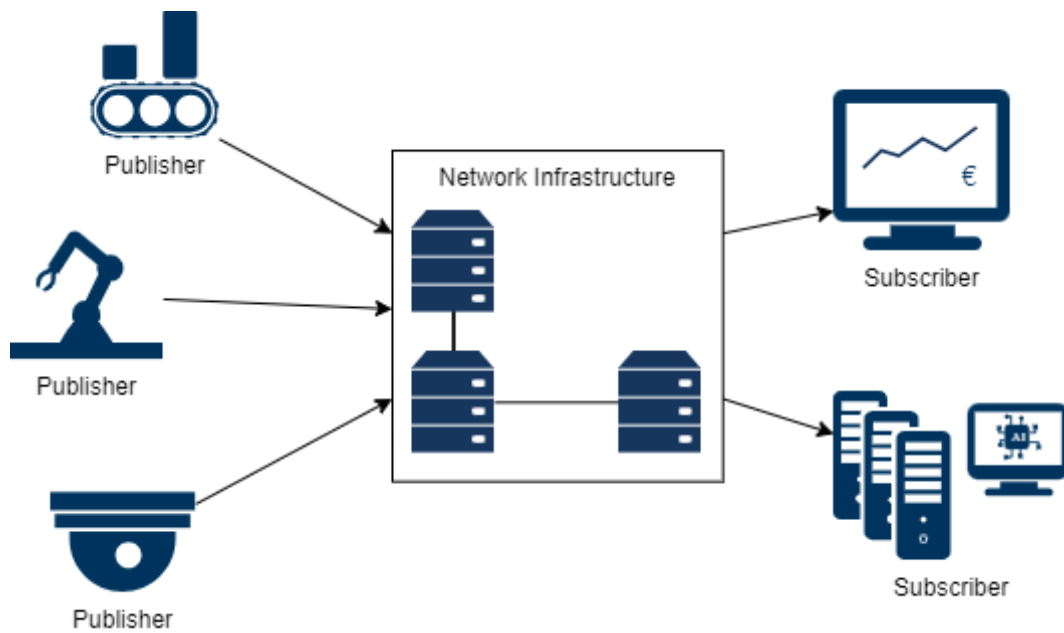


Figure 9: Broker-less PubSub

In broker-less communication, the UDP protocol is used for message transfer, and the UADP (binary encoded) protocol is used for message encoding. The multicasting technology allows for one-to-many and many-to-one communication. The binary-based encoding ensures that this communication is lightweight.

In broker-based communication, the MQTT protocol is the most commonly used for message transfer, although AMQP is also possible. However, MQTT is widely adopted in the industry, so the MOM is often an MQTT broker. It supports message encodings in human-readable JSON format and UADP.

3 MQTT

This chapter provides an in-depth exploration of MQTT, starting with an overview to establish its foundational concepts and principles. The subsequent section, "MQTT Essentials," dives into core aspects of the MQTT protocol, offering insights into its fundamental functionalities and mechanisms. Additionally, a dedicated subsection focuses on the essentials of MQTT version 5, highlighting the advancements and enhancements introduced in this latest version of the protocol.

3.1 Overview

MQTT is a messaging protocol developed by the Organization for the Advancement of Structured Information Standards (OASIS) [19]. MQTT is a widely used messaging protocol designed for IoT devices requiring efficient and reliable communication with minimal overhead. Due to these characteristics, MQTT is also suitable for machine-to-machine (M2M) environments [19]. MQTT operates on a publish-subscribe model in which data is sent from a publisher to a messaging broker, which delivers the message to one or multiple subscribers. MQTT is known for its low bandwidth usage, minimal power consumption, and scalability, making it a popular choice for IoT applications.

The protocol's lightweight nature is attributed to its simple header structure and use of the TCP/IP protocol. Offering three levels of Quality of Service (QoS), MQTT ensures reliable message delivery, making it suitable for real-time applications. It supports bi-directional communication, enabling easy broadcast of messages to groups of devices. MQTT also simplifies message confidentiality by employing TLS encryption and client identity verification through authentication protocols like OAuth.

Widely adopted across industries such as automotive, manufacturing, and oil and gas, MQTT benefits from extensive support through a vast ecosystem of libraries and tools, facilitating easy implementation and usage. Its main components—publisher, broker, and subscriber—will be explained further in the following chapter, along with essential MQTT features.

3.2 MQTT Essentials

An MQTT publisher refers to a device or application responsible for transmitting messages to an MQTT broker on a designated topic [20]. The payload, or data being sent, is created by the MQTT publisher and then published on the specific topic within the broker. Publishers can be a variety of sources, including sensors, machines, applications, or any device that needs to send data to other devices or applications within the MQTT network. Publishers can send messages to multiple topics, while the same topic can also have multiple publishers.

An MQTT broker is a messaging server functioning as a central hub for routing messages between publishers and subscribers in the MQTT communication protocol [20]. When a publisher sends a message, the broker receives it and forwards it to all subscribers registered to the corresponding topic. The broker manages the message

flow, ensuring their delivery to the intended recipients, while also providing additional services such as security, message persistence, and protocol translation.

An MQTT subscriber is a client that connects to an MQTT broker and subscribes to one or more topics to receive messages published on those topics [20]. When a message is published on a subscribed topic, the broker forwards it to all subscribers registered to that topic. To ensure reliable message delivery, subscribers can specify quality of service (QoS) levels. MQTT subscribers can be implemented in different ways, including as standalone applications, embedded devices, or as parts of larger systems.

Overall, the concept of MQTT communication can be easily understood through an analogy with TV or radio. Just like a TV channel that broadcasts a show to a TV station, in MQTT, a publisher sends data to an MQTT broker. The TV station acts as the broker in this analogy. Similarly, viewers tune in to a specific channel to watch their desired program. In MQTT, channels are referred to as topics, the TV program becomes the payload, and the viewer represents a subscriber who is interested in a particular topic. The analogy also highlights the fact that the publisher and subscriber do not need to have a direct connection with each other.

3.2.1 MQTT topic structure

MQTT topics are used to identify messages and route them to the appropriate subscribers [20]. Topics are strings that are organized hierarchically, with levels separated by forward slashes. Wildcards can be used in MQTT topics to represent groups of topics, providing a flexible and powerful way of subscribing to messages. The "#" character is a wildcard that matches any number of levels in the topic hierarchy, while the "+" character is a wildcard that matches exactly one level in the topic hierarchy. Topics starting with a "\$" symbol are reserved for internal statistics of the MQTT broker and are not part of the subscription with the multi-level wildcard as a topic.

There is progress to be made in advancing the topic structure. For example, Sparkplug is a standard that seeks to extend the MQTT protocol towards a more standardized solution. One aspect is the topic tree, for which they provide rules on how to construct it. The specification standardizes the topic as follows [21]:

```
namespace/group_id/message_type/ edge_node_id/[device_id]
```

Namespace and message_type are data that users cannot modify. The group_id, edge_node_id, and device_id can be configured by users, enabling data grouping at multiple levels. This standardized tree model also facilitates auto-recognition for subscribers, as they can assume certain levels from the topic.

Unified Namespace (UNS) represents another strategy for structuring the topic tree. In UNS, the concept involves constructing data with specific hierarchy levels and then aligning the topic tree to reflect these data hierarchy levels. This approach is more of an ideology than a solution, with the basic idea represented by the following structure [22]:

Enterprise / Site / Area / Line / Cell

OPC UA also provides guidance for constructing topics. While all the topic levels are detailed later in the work, the structure has some similarities to Sparkplug:

```
<Prefix>/<Encoding>/<MqttMessageType>/  
<PublisherId>/[<WriterGroup>/<DataSetWriter>]
```

Encoding, MqttMessageType, and PublisherId provide certain data and cannot be affected by the user. However, WriterGroup and DataSetWriter can be user-defined, allowing users to structure the messages.

3.2.2 MQTT features

MQTT Quality of Service (QoS) defines the level of assurance for message delivery between the MQTT publisher and subscriber [20]. There are three levels of QoS: QoS 0, QoS 1, and QoS 2 [20]. QoS 0 provides at most one delivery attempt for a message and does not guarantee message delivery to the subscriber. QoS 1 ensures at least one delivery attempt for a message and guarantees that the message will be delivered to the subscriber at least once, though duplicates may occur. QoS 2 provides exactly one delivery attempt for a message and guarantees delivery to the subscriber exactly once but with higher overhead.

Retained messages are special messages that brokers save and send to any new subscriber that subscribes to the corresponding topic [20]. When a publisher sends a message with the retained flag set to true, the broker stores the message and delivers it to new subscribers who later subscribe to the topic. The MQTT Last Will and Testament (LWT) feature allows a client to specify a message that the broker will send if the client disconnects from the broker ungracefully. MQTT's keep-alive feature ensures the continuity of communication between an MQTT client and broker by requiring the client to periodically send a message to the broker indicating that it is still active and connected. MQTT client's take-over feature is a mechanism that allows a new client to take over the connection and subscriptions of a previously connected client with the same identifier.

3.2.3 MQTT 5 features

MQTT's newest version, MQTT 5, was released in 2019. It is the next version after MQTT 3.1.1. While MQTT 5 maintains the fundamental architecture of MQTT, it introduces new features aimed at enhancing usability. The primary objectives for the development of MQTT 5 were [23]:

- Enhancements for scalability and large-scale systems
- Improved error reporting
- Extensibility mechanisms including user properties

- Performance improvements and support for small clients

MQTT preferred to add features wanted by the long-term users, without increasing overhead or decreasing ease of use and to improve performance and scalability without adding unnecessary complexity [20].

New and improved reason codes can now be used for a wider range of headers. For instance, a broker can send a response to a client's CONNECT packet with a CONNACK packet with pre-defined headers. These headers can be used to inform the client, which features are supported by the broker. This enables subscribers or publishers to inquire about the features offered by the broker without the need for design and configuration beforehand.

MQTT 5 has improved its authentication mechanism by introducing challenge-response communication and enabling password-only authentication. With this, the usage of tokens has become possible, without requiring a username, as was the case previously. In MQTT 5, a clean session is now referred to as a clean start. The session expiry time has also been added, along with the clean start. [20]

With the introduction of negative acknowledgment, MQTT 5 has enhanced the debugging and logging capabilities of the communication [20]. In practice, this means that the broker can now send a message together with an acknowledgment indication that the communication is not functioning correctly. Previously, the connection was simply rejected, leaving the client unaware of the reason for the rejection.

MQTT 5 introduced custom headers to the messages, similar to what HTTP has. These headers are named UserProperties which consist of key-value pairs. As long as the maximum message size is not exceeded, users can add as many UserProperties as they want. This feature makes it simple to add metadata to messages and was highly requested by MQTT version 3 users. [20]

Finally, MQTT 5 added an indicator for the payload format. It is divided into two parts: Payload Format Indicator and Content-Type [20]. When the Payload Format Indicator value is set to 0, it means that the payload is an "unspecified byte stream". On the other hand, when it is set to 1, it indicates that the payload is "UTF-8 encoded". The default value is set to 0. If the Payload Format Indicator is set to 1, a Content-Type value is expected, which can be any UTF-8 string that describes the content.

To summarize, MQTT is a messaging protocol known for its efficiency, reliability, and lightweight nature. It is particularly useful for IoT devices that require minimal code. MQTT has been widely adopted and supported, making it a popular choice across various industries. However, it is important to note that the specification does not include definitions for payload, context, and connection monitoring.

4 OPC UA PubSub

This chapter focuses on OPC UA PubSub, starting with an overview to provide context. Following this, it introduces the PubSub information model. A dedicated section then explores the topic structure within OPC UA PubSub, outlining its structure. The chapter further examines the DataSetMessage, covering both the VariableDataSet and EventDataSet. Additionally, a detailed analysis of the message header content is provided to enhance understanding. Finally, the chapter concludes with an exploration of discovery messages, with particular focuses on the DataSetMetaData and its significance in system configuration and operation.

4.1 Overview

OPC UA PubSub is a part of the OPC UA specification that declares a publish-subscribe architecture to the OPC UA ecosystem. OPC UA PubSub is not restricted to any specific communication protocol. The primary concept is that it operates as a publish-subscribe architecture. Thus, OPC UA PubSub supports communication protocols such as UDP and MQTT [11]. UDP messaging facilitates binary encoded messages known as UADP, while MQTT supports both UADP and JSON formats. JSON is a human-readable syntax that uses key-value pairs.

It is crucial to understand that OPC UA and MQTT are not equivalent. MQTT is a communication protocol that is used in OPC UA PubSub, while OPC UA is a communication architecture. There have been some misleading comparisons made between OPC UA and MQTT, which should be avoided.

It is important to distinguish PubSub communication from the subscription service implemented in the client-server communication model. In the client-server communication model, a subscription service is declared where the client requests updates from the server for specific data. The main difference between the subscription service and PubSub is the mechanism used. In the client-server subscription service, the session must be active, and the client and server are coupled.

In the automation industry, PubSub is targeted toward specific parts of the architecture. The following Figure 10 illustrates the use cases targeted with PubSub.

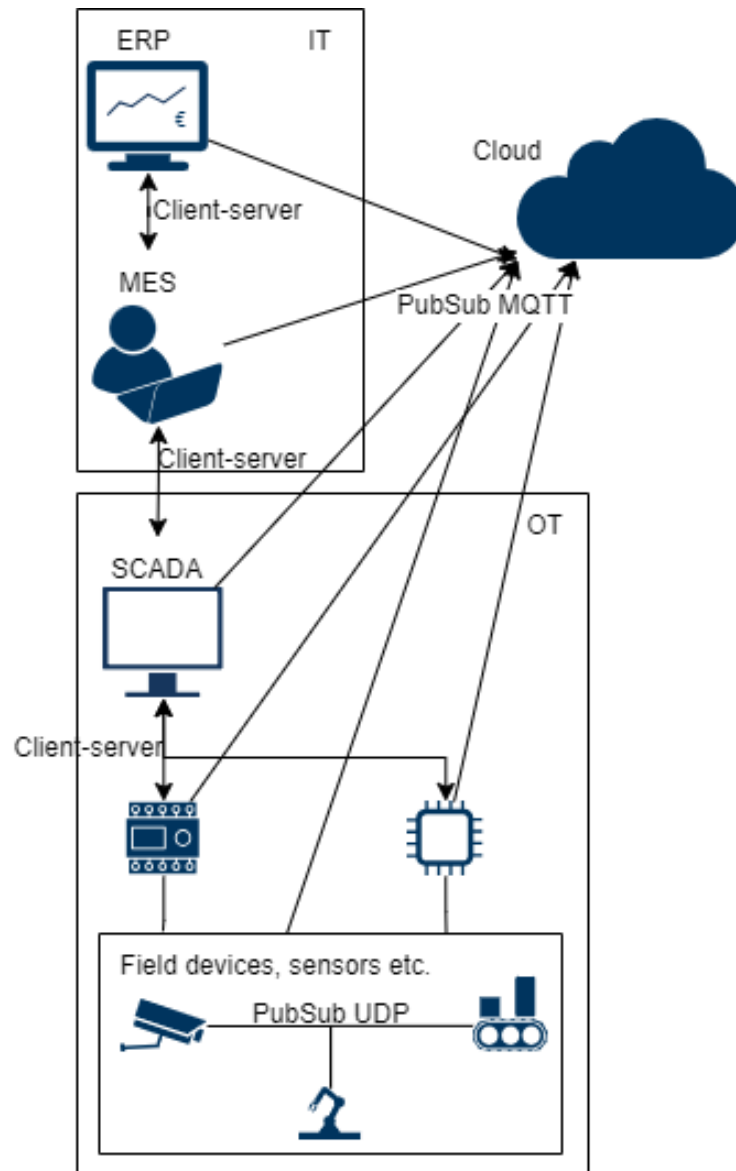


Figure 10: PubSub in automation industry architecture.

The PubSub architecture is designed to make OPC UA systems more scalable and efficient in sending messages. PubSub complements client-server communication, allowing OPC UA to cover a wider range of use cases. It is important to note that PubSub communication is not intended to replace client-server communication, but rather to extend the capabilities of the OPC UA system [11]. The next Figure 11 demonstrates the architectural change of industrial automation, where products and devices, are all communicating with each other.

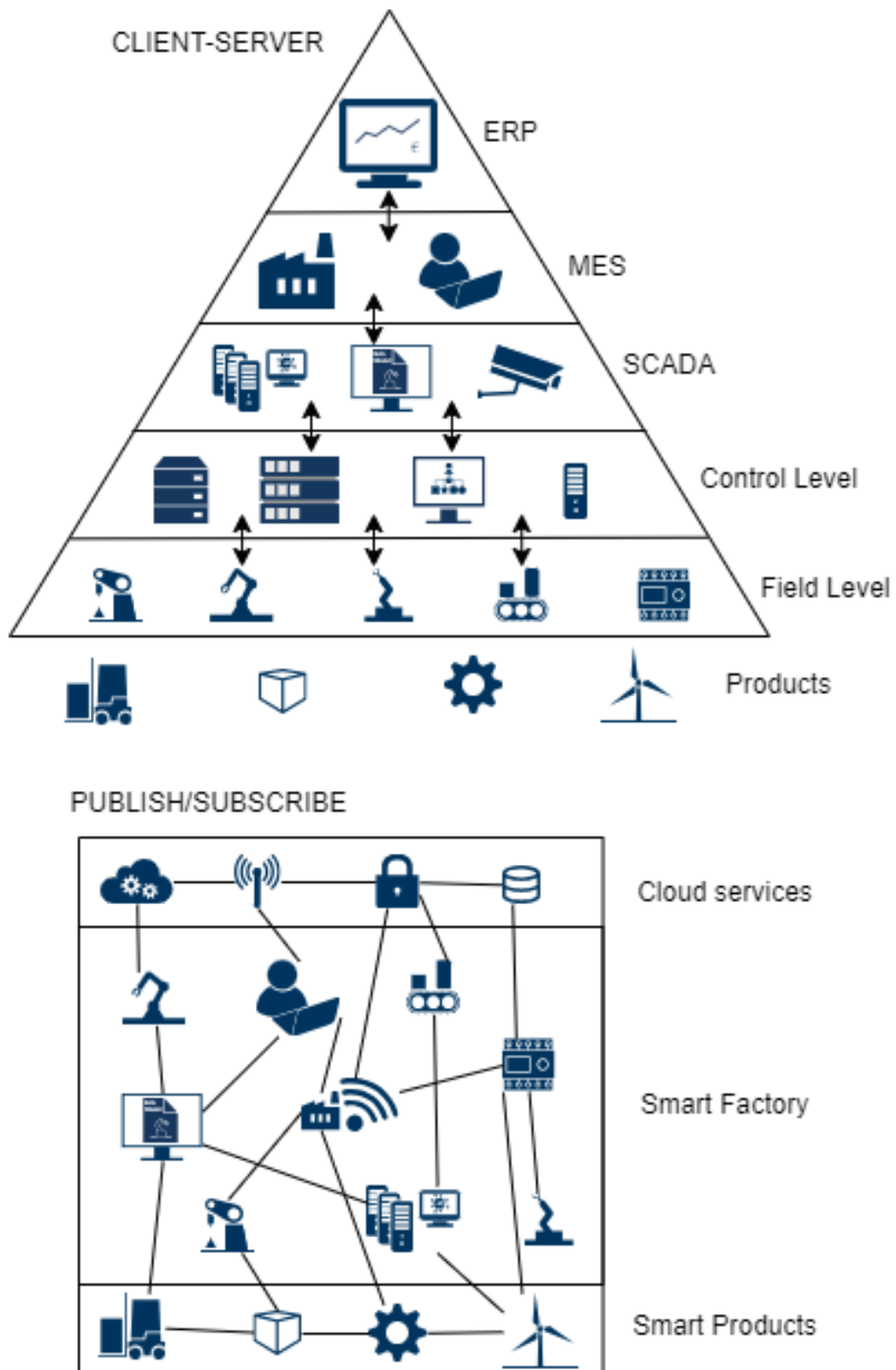


Figure 11: Overview of the architectural change in industrial communication.

OPC UA security architecture relies on the MQTT client ID, username, and

password. Additionally, it has a Security Key Service (SKS) specification that provides keys for message security. These keys can be used by the publisher to sign and encrypt NetworkMessages. The subscriber, on the other hand, uses the SKS to verify the signature of NetworkMessages and decrypt them.

It is worth noting that the OPC UA PubSub specification is a relatively new feature in OPC UA and is continuously evolving. The specification is being improved and supplemented all the time. As mentioned earlier, PubSub is not limited to MQTT, even though this work only focuses on the PubSub over MQTT of the specification.

4.2 PubSub information model

The OPC UA PubSub architecture comprises three key components: Publisher, Subscriber, and Message Oriented Middleware (MOM). Publishers are responsible for transmitting messages to the Message Oriented Middleware, without any prior knowledge of whether there are any subscribers interested in receiving the messages. Similarly, Subscribers request to receive messages on specific topics, without any knowledge of the message's source. Message Oriented Middleware serves as the intermediary component that manages message delivery and queuing within the system.

The information exchanged between systems is referred to as a DataSet, and its structure and additional metadata are described by the DataSetMetaData. In the publisher, the selection of data for the DataSet is called PublishedDataSet, which may comprise variable values or event fields.

A DataSetWriter in the publisher generates DataSet messages either periodically for variables or when an event occurs if the DataSet is event-based. Multiple DataSet messages may be grouped into a NetworkMessage, which is then transmitted from the publisher to the MOM to a specific topic defined by the DataSetWriter.

Subscribers filter or select the DataSet messages they want to receive. In UDP communication, this selection is usually based on identification information for the publisher and DataSetWriter within the NetworkMessage. In MQTT, subscribers typically subscribe to an MQTT topic to receive DataSet messages from the publisher.

The following Figure 12 illustrates, how the NetworkMessage is constructed.

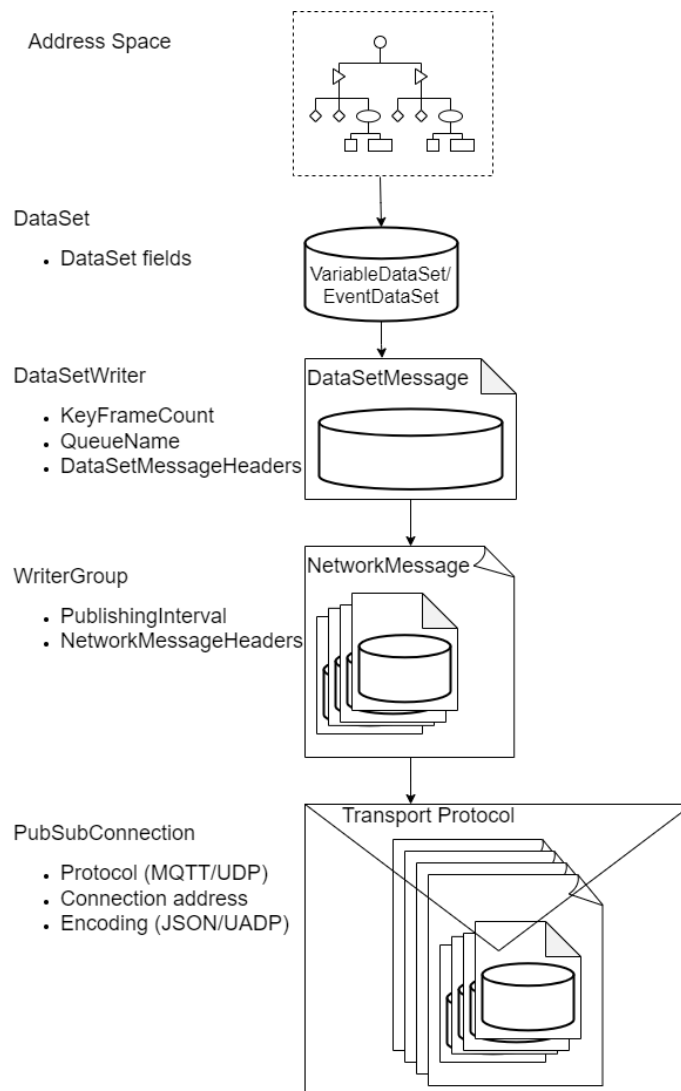


Figure 12: PubSub NetworkMessage structure and key configuration parameters.

4.3 Topic structure in OPC UA PubSub

Before the release of version 1.05.03, the PubSub specification lacked detailed instructions on how to use MQTT topics. However, the new PubSub specification includes a recommendation for the MQTT topic names:

$$\langle \text{Prefix} \rangle / \langle \text{Encoding} \rangle / \langle \text{MqttMessageType} \rangle / \langle \text{PublisherId} \rangle / [\langle \text{WriterGroup} \rangle / \langle \text{DataSetWriter} \rangle]$$

These levels come with default values or options to describe the message. The following Table 3 outlines these levels:

Table 3: The default topic levels defined in PubSub specification. [11]

Topic Level	Description	Value
<Prefix>	Value that can describe the wider context. For example, for a location the value could be a country or a smaller entity of the system such as factory	"opcua" is the default value. It can also be divided into multiple levels. It's important to note that this impacts the topic level structure, as the default levels do not match anymore.
<Encoding>	Which encoding is used	JSON or UADP
<MqttMessageType>	Pubsub defines six (6) message types.	"data", "metadata", "application", "endpoints", "status", and "connection".
<PublisherId>	Unique in the context of the prefix to identify a publisher	Can be UInteger and String. Zero UInteger and null or empty String values are not acceptable as PublisherIds.
<WriterGroup>	The name of a Writer-Group	The name identifying a Writer-Group within the Publisher.
<DataSetWriter>	The name of a DataSetWriter	The name identifying a DataSetWriter within the WriterGroup.

MessageTypes can be divided into two groups [11]:

- DataSetMessage: Used to share process and application data from the Publisher.
- Discovery messages: Comprising five other MessageTypes, these are employed to provide additional information not included in the DataSetMessages.

This thesis focuses on the two MqttMessageTypes: DataSetMessage and DataSet-MetaDataMessge. The latter is one of the types included in Discovery messages.

4.4 DataSetMessage

A DataSetMessage is created by a DataSetWriter and contains a collection of key-value pairs. It can include data related to variables or events. Fully explained DataSetMessage can be found in Appendix A.

4.4.1 VariableDataSet

Variable Nodes in DataSetMessages are sent cyclically with a parameter called Publishing interval that specifies how frequently the message is transmitted. In

VariableDataSets, the concepts of key frame and delta frame are important. A delta frame contains key/value pairs only for the values that have changed since the last transmission. Delta frames are generated at most once per interval. The key frame is a message that contains all the key/value pairs from the VariableDataSet. Keyframes are sent according to a parameter called KeyFrameCount, which is a multiplier for the Publishing interval. For instance, if the KeyFrameCount is 3, it means that after two delta frames, the third frame will be the key frame. The name of a DataSet field does not have any further instructions on how it should be formed, but it must be unique in the DataSet. [11]

4.4.2 EventDataSet

EventDataSet is used to create acyclic messages. These messages do not have a Keyframe or Deltaframe, so the KeyFrameCount for EventDataSets is always 0. Additionally, multiple event DataSetMessages can be created within one publishing interval, and these messages are then published at the interval. EventDataSet provides information on the properties of the events that occurred. [11]

4.5 Discovery messages

Discovery messages are used to provide additional information about the publisher application, which is not included in the DataSetMessage. The Retain flag should be true for all Discovery messages [11].

- DataSetMetaData is a required message type that defines the meaning of a DataSetMessage.
- ApplicationDescription describes OPC UA application and publisher capabilities. This message is optional.
- ServerEndpoints lists OPC UA endpoints of the OPC UA server, in case the Publisher is also a Server. This message is optional.
- Status. When a publisher connects to the MQTT broker, it must register a Status message. This message can either be cyclic, where the status is updated periodically, or acyclic, which means that the message is registered as an MQTT Will message to the MQTT broker, and it is sent after the publisher disconnects from the MQTT broker. It is mandatory for the Publisher to register the Status message.
- PubSubConnection contains configuration information for publisher's PubSubConnections. This message type is mandatory.

Let's take a closer look at the structure of the DataSetMetaData message. These messages contain information about the types and models used.

4.5.1 DataSetMetaData

The DataSetMetaData describes the structure of messages, enabling subscribers to interpret Data messages. It includes crucial details about DataTypes for the values transmitted. Additionally, DataSetMetaData messages may contain static information, such as engineering units, which remain unchanged over time.

The message is constructed with nested objects containing key-value pairs. To highlight a few headers, the message type can be identified from the following header:

```
"MessageType": "ua-metadata"
```

This can be an important header for filtering DataSetMetaData messages from others, especially when default topics are not used. In the default topic structure, messages can be separated by the MqttMessageType level.

DataSetMetaData should include a list of namespaces if the DataSet contains data from namespaces other than the basic OPC UA namespaces. The header should appear as follows:

```
Namespaces": [  
    "urn:DEMO-5:UA Sample Server",  
    "http://test.org/UA/Data/"  
]
```

The complete list of fields in DataSetMetaData can be found in Appendix B.

DataSetMessages can also be configured based on DataSetClass, which could be a globally described message structure. These DataSetClasses are identified by DataSetClassId values. Subscribers can use DataSetClassId to filter specific messages.

4.6 Message header content

Additionally, messages can have headers that contain data about the message and publisher. A JsonNetworkMessageContentMask defines which kind of information is included in the messages. This allows users to optimize between the message size and information provided for the subscribers. The JsonNetworkMessageContentMask is explained in the following list.

NetworkMessageHeader:

MessageId and MessageType are mandatory fields. If NetworkMessageHeader is true, the Messages field contains DataSetMessages. PublisherId, Writer-GroupName, and DataSetClassId are added if their respective flags are true.

DataSetMessageHeader:

Whether the DataSetMessages can have additional headers. The additional DataSetMessageHeaders are listed in JsonDataSetMessageContentMask.

SingleDataSetMessage:

Only a single DataSetMessage is included in the NetworkMessage. Alternatively, the NetworkMessage could contain a list of DataSetMessages.

PublisherId:

PublisherId is included in the NetworkMessage. This flag overrides the PublisherId flag in the JsonDataSetMessageContentMask, ensuring that the PublisherId is present only in the NetworkMessage.

DataSetClassId:

Whether the DataSetClassId is included in the NetworkMessage.

ReplyTo:

Reserved.

WriterGroupName:

Whether the WriterGroupName is included in the NetworkMessage. Similar to the PublisherId, this overrides the WriterGroupName flag of the JsonDataSetMessageContentMask.

If the flag for the DataSetMessageHeader is set to true, the JsonDataSetMessageContentMask adds fields to the DataSetMessage. The correct contents are described later in a JSON object that shows the message format.

DataSetWriterId:

Add DataSetWriterId to the message.

MetaDataVersion:

Add MetaDataVersion to the message.

SequenceNumber:

Add SequenceNumber to the message.

Timestamp:

Add Timestamp to the message.

Status:

Add Status to the message.

MessageType:

Add MessageType to the message. Note that this is different from the MessageType in the NetworkMessage.

DataSetWriterName:

Add DataSetWriterName to the message.

ReversibleFieldEncoding:

The DataSetMessage fields are encoded using reversible JSON encoding, while non-reversible JSON encoding is used in other cases.

PublisherId:

Include PublisherId in the message. If JsonNetworkMessageContentMask already contains PublisherId, this shall be skipped.

WriterGroupName:

Include `WriterGroupName` in the message. If `JsonNetworkMessageContentMask` already contains `WriterGroupName`, this shall be skipped.

MinorVersion:

Add `MinorVersion` to the message.

Finally, there is a third content mask `DataSetFieldContentMask`, which allows the addition of additional headers to the actual data fields.

StatusCode:

Show the status code with the value.

SourceTimestamp:

Add `SourceTimestamp` to the field content.

ServerTimestamp:

Add `ServerTimestamp` to the field content.

SourcePicoSeconds:

Add `SourcePicoSeconds` to the field content. This flag is ignored if the `SourceTimestamp` flag is not set.

ServerPicoSeconds:

Add `ServerPicoSeconds` to the field content. This flag is ignored if the `ServerTimestamp` flag is not set.

RawData:

If set to `True`, all others are ignored and the values are shown as key-value pairs.

Depending on the values of the `DataSetFieldContentMask`, the data can be presented in three different ways.

- **RawData:** This field displays only the value if `RawData` is selected. All other flags are ignored.
- **Variant:** The field contains the `Value` or `StatusCode`. This is used if none of the `DataSetFieldContentMask` options are selected.
- **DataValue:** The field is represented as a `DataValue`, including the value, timestamp, and status code.

Data messages contain headers with specific names for the fields, followed by user-defined names for the fields. For detailed descriptions of the fields in JSON Data messages and the corresponding values, please refer to Appendix A.

4.6.1 Header layouts

The PubSub specification aims to simplify message structure by providing three distinct header layouts with predefined ContentMasks. This design approach combines mandatory flags with configurable options, resulting in a wide range of message structures. However, this approach can lead to some complexity, as the utilization of the same layout may result in diverse-looking messages due to the extensive variety offered by configurable options.

The first layout is designed for scenarios that require a small message size. It is typically used when an OPC UA Publisher communicates with IT applications that lack OPC UA knowledge. In such cases, there is no need for OPC UA-specific information or headers. In this layout, the `SingleDataSetMessage` flag is set to true.

The second layout is used when a single DataSet is linked to a specific topic. Similar to the first layout, the message consumer may not have OPC UA awareness, but can still make use of the OPC UA header information for message processing. In this configuration, both the `DataSetMessageHeader` and `SingleDataSetMessage` flags are activated. The PubSub documentation provides additional customization options for `JsonDataSetMessageContentMask` and `DataSetFieldContentMask`.

The third layout is designed for situations where several data and event DataSets are published to a single topic. This layout provides a wide range of flag configuration options, but the main difference lies in the activation of the `NetworkMessageHeader` and `DataSetMessageHeader` flags while deactivating the `SingleDataSetMessage` flag. For further information on configuration parameters, please refer to the PubSub specification documentation.

5 Experimental PubSub setup

In the upcoming chapter, we will provide a detailed breakdown of the experimental setup, starting with an overview of its architecture. Each component will be explained, and the following applications will be showcased: Prosys OPC UA Simulation Server, UaModeler, Prosys OPC UA Forge, Mosquitto Broker, and Prosys OPC UA Browser. Following that, the Information Model, which demonstrates how data is organized in our system, will be declared with explanations and diagrams. Finally, we'll look into how OPC UA Forge data is configured, publishers are set up, and communication elements are defined. The primary aim of this chapter is to give you a clear picture of the setup.

5.1 Target of the PubSub configuration

The key consideration is how effectively the PubSub specification has been established to OPC UA specification. One of the most critical aspects of the OPC UA specification is the standardized information models that include all metadata related to the Nodes. The question is whether this information can be transferred efficiently over low-bandwidth communication channels, without knowledge of the other end of the communication. In this regard, we can examine the metadata messages that PubSub employs to determine how well they supplement the data messages.

It is interesting to observe how effectively the communication of information models can be ensured for all participants involved in the communication process. With a large amount of information models and continuous development, ensuring that every participant is updated at all times is a challenge. OPC UA addresses this challenge by employing namespaces and identifiers to uniquely identify information models.

Lastly, it's essential to assess how well the information can be reconstructed on the subscriber side in order to understand the source of the data. This is particularly important in OPC UA data modeling and harmonization. One question that arises is whether it's necessary to reconstruct the entire server where the messages are published, or if it's possible to identify a specific Node using its namespace and identifier. Another important consideration is how to access the namespace information of the server without having to connect to it.

It is worth noting that messages created by OPC UA publisher, can be received by any MQTT subscriber. Therefore, it is important to consider the challenges that arise from communicating with non-OPC UA devices.

5.2 Architecture of the setup

The experimental setup is done entirely on a single computer, and therefore, the connection performance or the effectiveness of the communication is not measured in detail. However, the focus is to analyze the information and message architecture. The setup consists of multiple software applications, all of which, except for Forge, have

free versions that can be installed and tried. Figure 13 below illustrates the structure of the experimental setup.

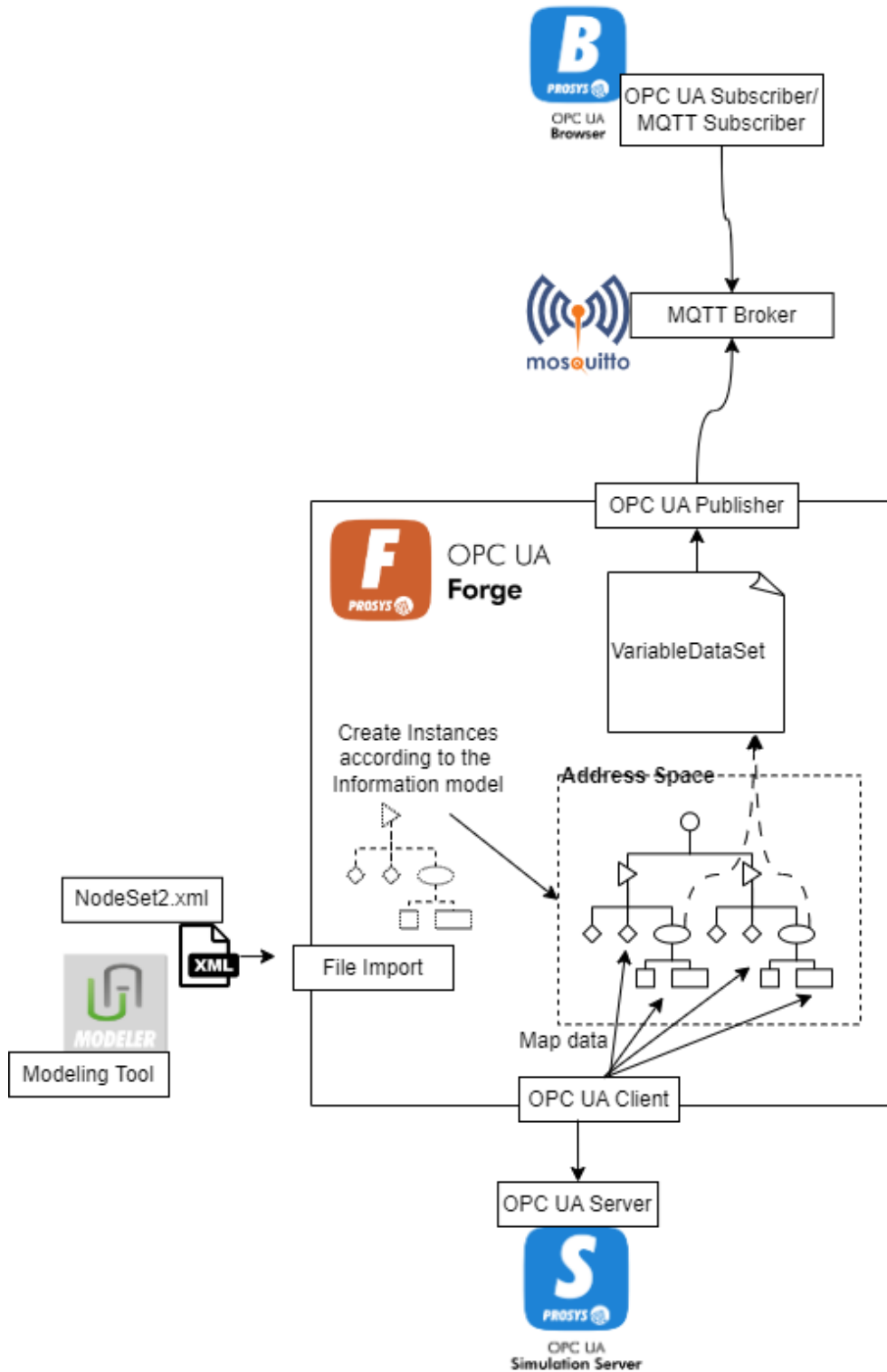


Figure 13: System architecture of the experimental setup.

This experiment demonstrates how modern automation data can be unified and OPC UA information models can be utilized. The Prosys OPC UA Forge is the central component of this experiment. It can aggregate OPC UA servers, map data between Nodes, and configure the OPC UA publisher. An information model is created using UaModeler and then imported to Forge using the NodeSet2.xml format. This enables the use of all the types declared in the information model for instance declaration. Forge is then connected to the Prosys OPC UA Simulation Server using a Client-Server connection to receive dynamic simulated data, which is then mapped to the instances in Forge. This way, the experiment can have changing values without real sensors. Lastly, the instances with values can be configured to be published using the MQTT protocol to Mosquitto broker. In this experiment, the messages are subscribed by the Prosys OPC UA Browser.

5.2.1 UaModeler

UaModeler is a desktop application developed by Unified Automation, a company that creates and distributes software and components for industrial automation [24][25]. UaModeler provides users with graphical design tools to build OPC UA Types and instances, allowing them to create and design the OPC UA Information model with ease. With the help of a graphical view, users can generate ready and error-free code for their OPC UA project, as well as the NodeSet2.xml file which can be used for importing information into an OPC UA server.

The OPC UA NodeSet2.xml files are an essential component of OPC UA. They enable OPC UA servers to import OPC UA information models to be used in them.

5.2.2 Prosys OPC UA Simulation Server

The Prosys OPC UA Simulation Server is a useful tool that can be used to test OPC UA client applications and to learn more about the OPC UA technology [26]. It has been developed by Prosys OPC Ltd, a company that specializes in developing OPC UA software and offers professional services for the automation industry [27].

With the Simulation Server, users can create an Address Space that includes simulated variables, import information models, configure PubSub communication, and take advantage of many other useful features. It is available as a free version, as well as a Professional version that includes additional features.

One of the best things about the Simulation Server is that it is user-friendly. All configurations can be easily made through a modern, intuitive interface that guides users through the process.

5.2.3 Prosys OPC UA Forge

Prosys OPC UA Forge is a recently released application by Prosys OPC Ltd, which provides modern OPC UA integration software on the edge [28]. The architecture that lies between operational technology and information technology, but still exists on the operational technology side, is what is meant by 'edge'. Forge offers several

features, including data formation, harmonization, reconstruction, logging, and server aggregation. Forge can connect to OPC UA servers through a client-server connection, and any OPC UA client can connect to Forge. Moreover, the software comes with Modbus connectivity and PubSub configurations. Unfortunately, Forge is commercial software, but it is available for free evaluation.

5.2.4 Mosquitto Broker

Mosquitto Broker is a lightweight open-source MQTT broker that supports protocol versions 5.0, 3.1.1, and 3.1 with high customization options [29].

5.2.5 Prosys OPC UA Browser

Prosys OPC Ltd has made a generic OPC UA client software called Prosys OPC UA Browser [30]. It is very simple and easy-to-use OPC UA client software that can be used to test OPC UA servers. The Browser does not only support Client-server connections but it allows connections to MQTT brokers to receive MQTT messages. The benefit is that the Browser is OPC UA compatible and it can parse the PubSub messages. Additionally, it can receive any custom MQTT messages.

5.3 Information model

A new information model was developed for this thesis to demonstrate the possibilities of OPC UA information modeling. The purpose was to experiment with how the information model could be configured for further usage. To illustrate this, a simple model was created with the theme of a smart home, which would be easy to understand and help evaluate the results of sending and analyzing messages.

When designing the model, various data types and structures were included to enhance its versatility. This included ReferenceTypes, basic variables, structured variables, arrays, and objects. By creating a hierarchy within the model, it was ensured that the MQTT messages could have a wide range of complex types that can be analyzed.

UaModeler was used to create the model. The model was initially designed as a diagram utilizing the graphical representations of the OPC UA specification, as shown in Figure 6 presented earlier in this thesis.

The following Figure 14 shows three (3) ObjectTypes that are sub-types of BaseObjectType.

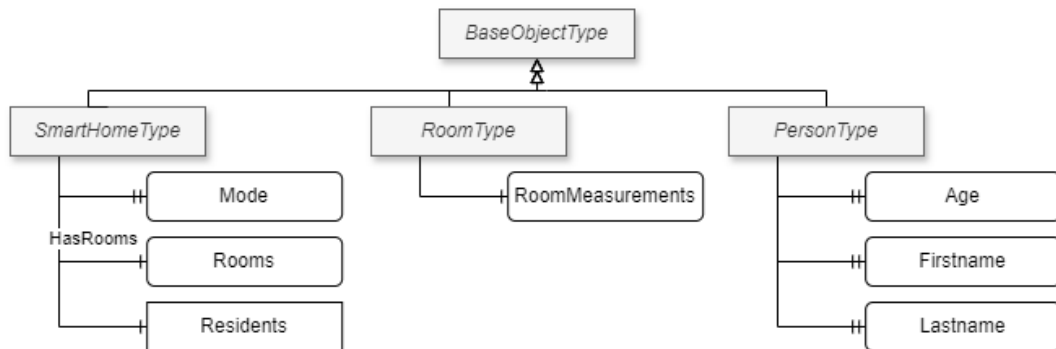


Figure 14: ObjectTypes of the created Information model.

The SmartHomeType is a representation of different home instances. You can have multiple home instances to represent different homes and keep all the information on specific homes under the SmartHomeType instance. In this model, the SmartHomeType has one property called Mode. The Mode property has a current state value in the form of an enumerator, such as Home, Away, and Night. The SmartHomeType also has one variable, Rooms, which is an array of strings that lists all the rooms connected to the SmartHomeType instance.

Additionally, there is one object called FolderType that groups every resident of the home. The residents can be set either as an ObjectType PersonType (presented in Figure 14) or as a structure VariableType PersonVariableType (presented in Figure 15). This was done to compare the benefits of using one or the other.

Moreover, the SmartHomeType is supposed to have references to one or many RoomType instances. These instances would represent each room of the home.

RoomType is used to group measurements from a specific room. In this case, one structured variable is used to represent three (3) measured values. The TypeDefinition of the variable is RoomMeasurementVariableType, which is presented in Figure 15.

The last ObjectType is PersonType, which defines data of a person. This is used to represent residents of a home and is connected to the Residents' object. The PersonType has three (3) properties: Age, Firstname, and Lastname.

This information model includes two (2) VariableTypes that are sub-types of BaseDataVariableType, as represented in the following Figure 15:

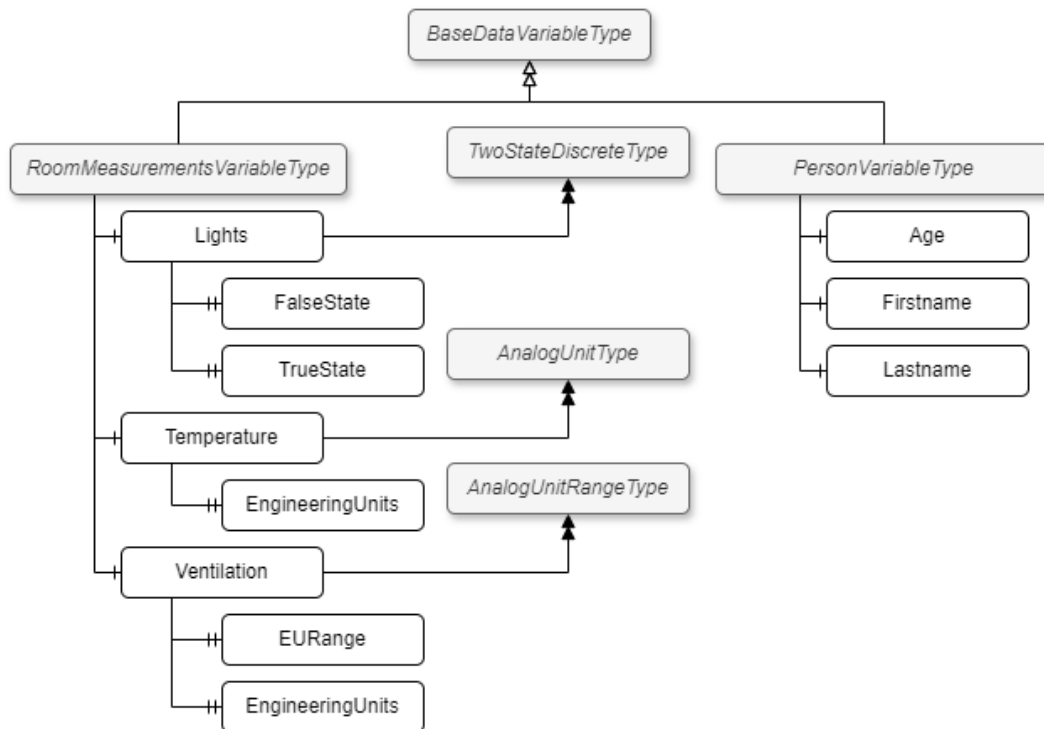


Figure 15: VariableTypes of the created Information model.

RoomMeasurementVariableType groups a set of measurement variables into the same structured variable. The variables under RoomMeasurementVariableType are not created from BasicDataVariableType; instead, they use TypeDefinitions to VariableTypes with mandatory properties. For instance, the Lights variable, which describes whether the lights are on or off in the room, has a TypeDefinition of TwoStateDiscreteType. This creates two properties as mandatory for the variable: TrueState and FalseState, which describe the two possible values. The Temperature variable is typed using AnalogUnitType, which makes the property EngineeringUnit mandatory. The last variable, Ventilation, represents the power of ventilation as a percentage value. Thus, an AnalogUnitRangeType is used, which requires the EURange and EngineeringUnit properties. With these properties, the value of the variable can be described to be between 0 and 100%, with the EngineeringUnit as percent (%).

The PersonVariableType is a VariableType that serves as an optional way to represent a person instance in the model. This VariableType is similar to the PersonType, which is created using ObjectType. All variables connected to the PersonVariableType are created using BaseDataVariableType, which means that they do not have any mandatory properties that describe the values further.

In order to test how the PubSub communication can forward model information, a new ReferenceType, named HasRooms, was created. Its main purpose is to be used between SmartHomeType and the Room variable. This ReferenceType is a sub-type of the HasComponent ReferenceType, as shown in the diagram in Figure 16.

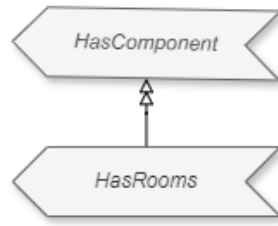


Figure 16: ReferenceType of the created Information model.

This chapter only described the Information model that was created, but it should be noted that the Address Space/instances that are going to be created into Forge using the above models will be described later.

5.4 Simulated variables

Simulated variables were created using Simulation Server. The benefit of using simulated variables is that the system can be tested with changing values, allowing the data flow to be observed and verified. In this test scenario, we can also verify how often messages are sent because, in the PubSub system, messages can be sent containing only changed values or every value. Hence, the simulated values allow us to easily analyze the actions in this regard. Table 4, presents a comprehensive list of the simulated variables used in the experimental setup, detailing their respective types and simulation parameters.

Table 4: Simulated Variables Overview

Name	Data Type	Value simulation
Bedroom/light	Boolean	Sawtooth (min: -2, max: 2, period: 10, time offset: 0)
Bedroom/temperature	Double	Sinusoid (min: 21, max: 27, period: 30, time offset: 0)
Bedroom/ventilation	Int16	Random (min: -2, max: 2)
Livingroom/light	Boolean	Sawtooth (min: -2, max: 2, period: 10, time offset: 0)
Livingroom/temperature	Double	Triangle (min: 19, max: 25, period: 120, time offset: 0)
Livingroom/ventilation	Int16	Random (min: 0, max: 100)

5.5 Configuring OPC UA Forge data

The process of configuring OPC UA Forge involves several steps. Initially, the connection to the Simulation Server was established using the client-server communication model. Next, the created information model was imported using the NodeSet2.xml

file, and instances were generated based on these models. Data was then mapped from the Simulation Server to these instances. Finally, a publisher was configured with various settings.

5.5.1 Connecting Forge to Simulation Server

The connection between Forge and the Simulation Server is established through a standard OPC UA client-server connection, with Forge acting as the client and the Simulation Server as the server. The default connection address for the Simulation Server is:

`opc.tcp://localhost:53530/OPCUA/SimulationServer`

The Address Space of the Simulation Server can be accessed from Forge, allowing the data retrieved from the Simulation Server to be mapped to different Nodes. The next step in configuring Forge involves importing the created information model and then creating instances using the imported types.

5.5.2 Address Space configuration

Forge application was selected for this experiment due to its ability to easily adapt information models, its user-friendly configuration interface, and its status as a state-of-the-art application for edge usage within the OPC UA environment. Forge allows users to import information models, and use them to implement instances. Additionally, Forge provides an interface for configuring PubSub communication, making it well-suited for observing how information models are integrated into MQTT messaging.

The importing process was initiated through a simple dialog, which allowed the created NodeSet2.xml file to be uploaded to Forge. The NodeSet2.xml, created with UaModeler, included only types. Upon completion of the process, a new namespace, `http://mastersthesis.org/SmartHome/`, was added to Forge's namespace table. The types could be now utilized in the Address Space to generate instances. The following instances were then created in the Address Space as outlined in the Figure 17 below:

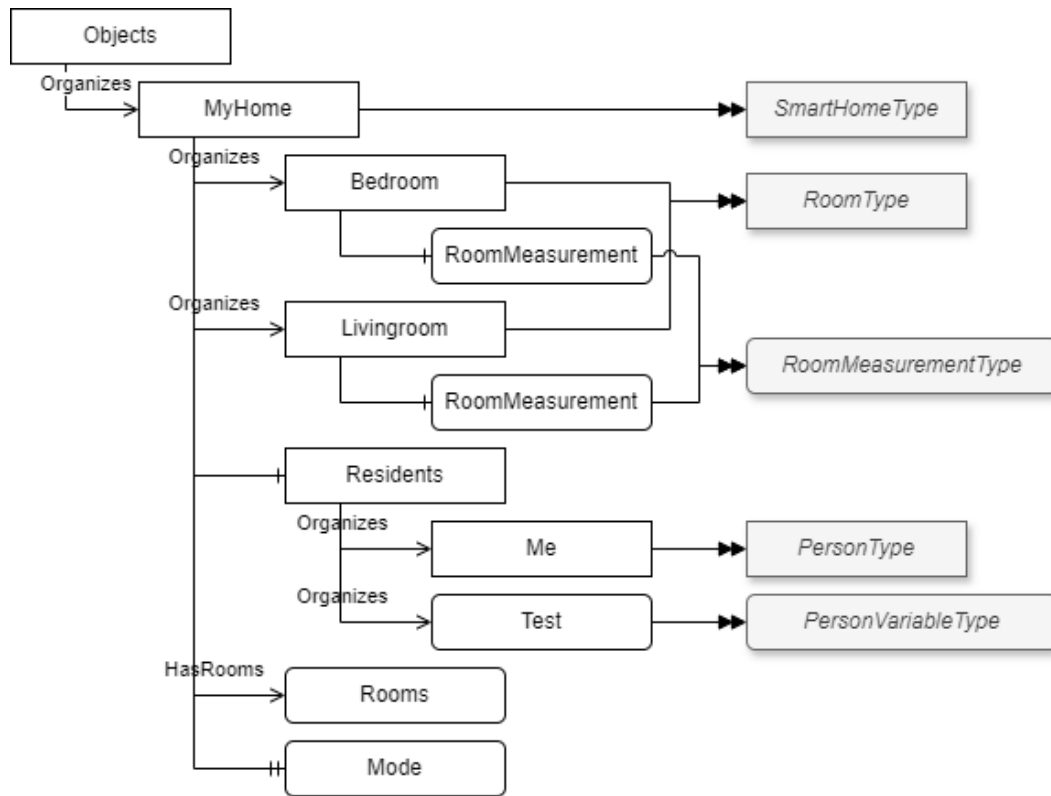


Figure 17: Diagram of the Address Space created.

An instance of the SmartHomeType named MyHome was initially created. This established a structure comprising objects such as Residents, variables like Rooms, and properties such as Mode. Then, two instances of RoomType were created under the MyHome object. Additionally, two persons' expressions were created within the Residents object each serving distinct purposes. One was created using the ObjectType PersonType, while the other by using PersonVariableVariableType. These two options can be compared in MQTT messaging, to see whether complex VariableType is better than ObjectType. The following figure, Figure 18, illustrates the user interface for configuring the Address Space within the OPC UA Forge application, showcasing the fields required to create new instances.

< New Node

Type Definition

Model type URI filter
http://mastersthesis.org/SmartHome/

Model type*
RoomType

Node Identity

Name*
Livingroom

Namespace *
[2] http://www.prosysopc.com/OPCUA/Forge

Id Type *
String

Identifier*
MyHome/Livingroom

Save Cancel

Figure 18: Configuring a new Node into the Address Space.

As seen from Figure 18 above, the user needs to define a TypeDefinition, name, and identifier for the new Node. Additionally, the user can select in which namespace the new Node will be added. Saving the form above results in the following structure:

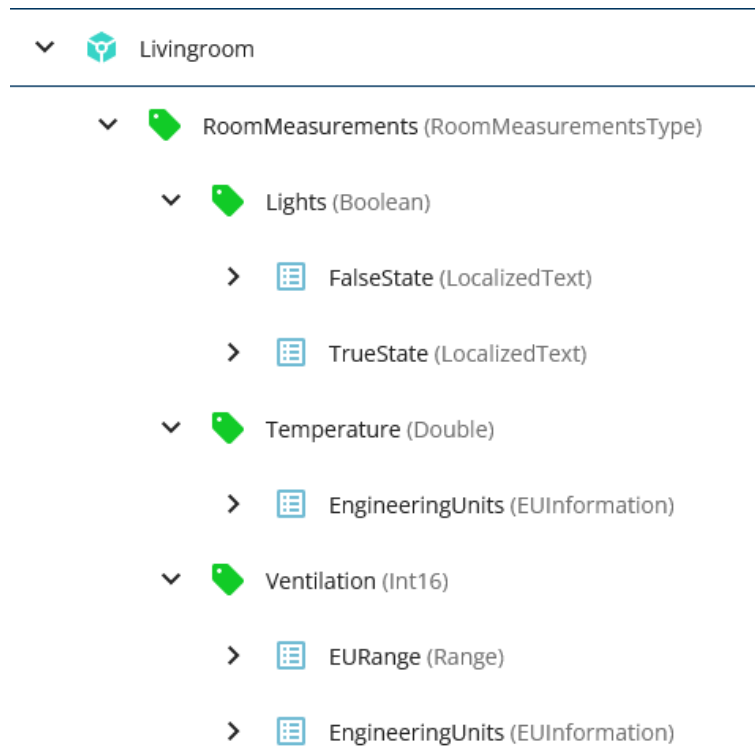


Figure 19: Instantiated type in the Address Space.

Forge allows users to map attributes from one Node to another. In this experiment, the mapping of value attributes from Nodes in the Simulation Server to Nodes created within Forge was done manually using Forge’s UI. Values without simulated options, such as Rooms and Mode, were input using the Browser application. For both person types, values were added using a static mapping feature, allowing static values to be saved to these Nodes. This enabled the completion of the Address Space by utilizing instances based on the information model and adjusting values in variable Nodes as required.

In this context, the functionality of Forge enables attributes from Nodes within a Simulation Server to be effectively associated with Nodes generated within Forge. This capability completes the Address Space, as it facilitates the utilization of instances derived from an information model and dynamic values within variable Nodes. Thereby enhancing the overall functionality of the system. The Figure 20 illustrates the mapping between the Simulation Server and the light Variable created in Forge, showcasing the user interface.

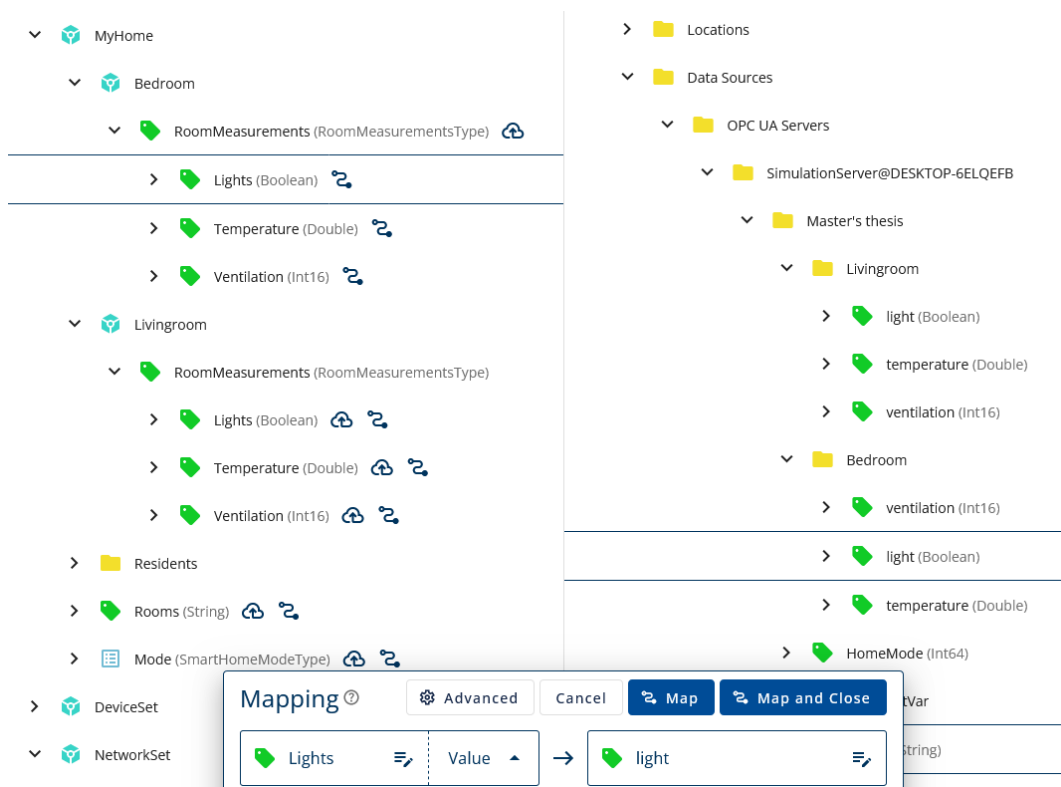


Figure 20: Mapping of attributes in Forge.

With the Address Space established, the next step involves configuring an OPC UA publisher responsible for publishing the variables from the model. Before declaring the Publisher, the MQTT broker configuration is introduced.

5.6 Mosquitto broker

This experiment used a locally installed Mosquitto broker, operating with default configurations. Mosquitto broker was installed as a Windows service, with the default port set to 1883.

5.7 Configuring Publishers to Forge

Forge features a module for configuring an OPC UA publisher. The focus here is on the possibility of sending the values of MyHome variables also with contextual information. The PubSub configurations have few places where the message content of the MQTT messages can be affected. The PubSub configurations provide a few options for customizing the content of MQTT messages. The first step was to create DataSets that gather the data from the Address Space.

In OPC UA, two types of DataSets can be created for publication. They have different characteristics, as one type is called VariableDataSet, which contains data from variables, while the other is EventDataSet, used to publish properties of an OPC UA event. In this experiment, both types of DataSets will be configured. The

VariableDataSet will be described in the following chapter, and then new events will also be configured in the Address Space using Forge's Event Generator module, followed by publishing those events with EventDataSet.

5.7.1 VariableDataSet

For this experimental setup, a VariableDataSet was created. The configuration process of the VariableDataSet has three strategies to configure the name of the DataSet fields in the MQTT messages. Additionally, the name of the DataSet fields can be customized freely. The strategies that Forge offers are:

- **BrowseName:** BrowseName is used to identify the Node from the MQTT message. ("RoomMeasurements")
- **BrowsePath:** BrowsePath without namespace index is used to identify the Node from the MQTT message. ("MyHome/Bedroom/RoomMeasurements")
- **ExpandedNodeId:** ExpandedNodeId is used to identify the Node from the MQTT message. ("nsu=http://www.prosysopc.com/OPCUA/Forge;s=MyHome/Bedroom/RoomMeasurements")

ExpandedNodeId was selected for the strategy since it was considered to have the most information on the model. Next, the Variables were selected to be included in the messages. The following Variables were added:

- RoomMeasurements variable from Bedroom
- Lights variable from RoomMeasurements of Livingroom
- Temperature variable from RoomMeasurements of Livingroom
- Ventilation variable from RoomMeasurements of Livingroom
- Mode property
- Rooms variable

These variables were selected to enable a comparison between simple variables and structured variables. Additionally, an array variable and an enumerator were included for analysis of the message content. Residents were excluded from the DataSets because the PersonType object could not be added directly to the DataSet. Each property would need to be added separately, which would not introduce any new aspects.

Additionally, the DataSets shown in Table 5 were created to support UNS topic strategy. These DataSets utilized BrowseName strategy for the names of the DataSet fields.

Table 5: DataSets for UNS topic structure.

DataSet	Variables
MyHome	Mode, Rooms
MyHomeBedroom	RoomMeasurements
MyHomeLivingroom	RoomMeasurements

5.7.2 EventDataSet

The configured model does not include any events. Therefore, the initial step in configuring the EventDataSet is to use Forge's Event Generator module, which enables the creation of BasicEventType events.

An event generator was used to create an event indicating when lights are turned on inside a room. The configuration in Forge appeared as follows:

The screenshot shows the configuration interface for an event named 'LightsOn'. The interface is divided into several sections:

- General settings:** The 'Name*' field is set to 'LightsOn'. The 'Event Type' dropdown is set to 'BasicEventType'.
- Event Property templates:** There are two templates defined:
 - #1:** The 'Event Property' is 'Message'. The 'Template *' is 'LightsOn-Event: Light state: "{msg}"'. The 'Detected variables' list contains 'msg'.
 - #2:** The 'Event Property' is 'Severity'. The 'Template *' is '100'. The 'Detected variables' list is empty.

At the bottom right, there are 'Save' and 'Cancel' buttons.

Figure 21: Re-usable event template.

These event templates are re-usable for Event generators and multiple properties can be configured. Message property has one placeholder (msg) for a variable. Also, the Severity event property is configured to be 100. The Figure 22 below illustrates how a new event is added for an Object. The Event Generator generates a new event for the Livingroom object. The template includes one placeholder for a value, which

is filled with the boolean value from the Lights variable. The Lights variable is also used for the trigger, and the event occurs when the value equals true.

The screenshot displays the configuration interface for an Event Generator, organized into three main sections:

- General settings:** Contains three input fields: 'Name' with the value 'LightsOnLivingroom', 'Source NodeId' with the value 'nsu=http://www.prosysopc.com/OPCUA/Forge;s=MyHome/Livingroom', and a 'Template' dropdown menu set to 'LightsOn'.
- Template Variables:** Shows a single variable '#1' named 'msg'. Its 'Source NodeId' is 'nsu=http://www.prosysopc.com/OPCUA/Forge;s=MyHome/Livingroom/RoomMeasurements/Lights/TrueState'.
- Generation Triggers:** Contains one trigger '#1' with the following settings:
 - 'Trigger NodeId*': 'nsu=http://www.prosysopc.com/OPCUA/Forge;s=MyHome/Livingroom/RoomMeasurements/Lights'
 - 'Monitoring Mode*': 'Subscription' (dropdown)
 - 'Interval (ms)*': '500' (input field)
 - 'Trigger conditions':
 - 'Condition type*': 'Equals' (dropdown)
 - 'Comparison value*': 'true' (input field)

Figure 22: Event Generator configuration.

These events could be configured to be sent through PubSub communication using EventDataSet. Users can select which properties of the event are included in the message. In this configuration, Severity, Message, EventType, SourceNode, and SourceName were selected. EventType is added to see how it is presented and if it could help in the information model recognition. Then SourceNode and SourceName are used to see how the Node can be identified with those properties. Severity is just an addition.

5.7.3 Configure a connection to MQTT broker

A basic MQTT connection was selected and JSON encoding was used since binary-encoded messages are not human-readable without binary decoding. To set this connection the following configurations were set as shown in Figure 23:

InformationModeling
MQTT://LOCALHOST:1883
✎ ✕

General settings
ENABLED

Name

Publisher ID

Connection address settings

Protocol*

Hostname / IP*

Port*

MQTT settings

Client ID*

MQTT payload format

Payload Format

Figure 23: PubSub over MQTT connection configurations.

5.7.4 Configure WriterGroups

The connection created needs WriterGroups. A WriterGroup determines the publishing interval and the used JsonNetworkMessageContentMask. In this setup, two WriterGroups were created. One for using the default OPC UA topic structure and another to use a different strategy. Both WriterGroups used three (3) seconds publishing interval and the following flags were included:

- NetworkMessageHeader
- DataSetMessageHeader
- PublisherId

This JsonNetworkMessageContentMask follows the third layout.

5.7.5 Configure DataSetWriters

Finally, the DataSets are configured to be sent by DataSetWriters. A few parameters need to be selected for the DataSetWriters, such as the KeyFrameCount. Although it does not affect the message content, it is important to notice that when using the

EventDataSet, the KeyFrameCount should be set to 0, meaning that there are no cyclic messages. On the other hand, when using the VariableDataSet, the KeyFrameCount should be set to 1 or greater, indicating how often a key frame message is sent, as mentioned previously. For VariableDataSets the KeyFrameCount was set to 3.

In this experiment, the following JsonDataSetMessageContentMask was configured:

- MessageType
- DataSetWriterId
- DataSetWriterName

The MessageType was selected because KeyFrameCount was not 1; thus, this header could easily differentiate between the key and delta frames. Then, DataSetWriterId and DataSetWriterName were selected to provide information about the writer.

DataSetFieldContentMask included only the StatusCode.

The last parameter configured was the topic. This experiment utilized two different topic structures. The other one followed the default topic structure outlined in the OPC UA specification [11].

```
opcua/json/metadata/urn:DESKTOP-6ELQEFB:  
OPCUA:Forge/MyHome/MyHomeExpandedNodeIdDataSet
```

and for the metadata

```
opcua/json/data/urn:DESKTOP-6ELQEFB:  
OPCUA:Forge/MyHome/MyHomeExpandedNodeIdDataSet
```

The alternative topic strategy involved using the Address Space hierarchy to formulate the topic structure, similar to the UNS topic strategy. At this stage, it was realized that variables needed their own DataSets and DataSetWriters to exist within their respective topics, reflecting the hierarchy. As a result, the topics were structured in the following example:

```
Objects/MyHome/Livingroom/RoomMeasurement
```

For this scenario, the metadata topic was identical to the data topic due to the topic strategy does not have a logic for metadata topics.

5.8 Subscribing to the topics

To receive the messages, Prosys OPC UA Browser was used. Connecting Prosys OPC UA Browser to the MQTT broker is a simple process that only requires the endpoint of the broker. Once a broker is identified using the provided endpoint, the Browser will ask for a topic filter, client ID, username, and password. Authentication via username and password is optional since the broker does not mandate it. The client

ID is pre-filled with a default value. In this case, the wildcard # was used as the topic filter since it subscribes to every message sent to the broker.

The Log view enables the reading of raw message content, which remains human-readable due to the use of JSON encoding. Any MQTT subscriber is capable of receiving these messages, facilitating communication from OPC UA to custom subscribers.

6 Results

In this chapter, the generated messages are analyzed and compared to each other. The main focus is to identify how much information there is about the model. First, the topic strategies are reviewed, and then the message structure and different DataSets.

6.1 Topic structure

In this experiment, two different topic structures were created, other one followed the OPC UA specification and the other UNS topic structure.

6.1.1 Default OPC UA topic structure

```
opcua /
  json /
    metadata /
      urn :DESKTOP-6ELQEFB :OPCUA : Forge /
        MyHome /
          MyHomeExpandedNodeIdDataSet
    data /
      urn :DESKTOP-6ELQEFB :OPCUA : Forge /
        MyHome /
          MyHomeExpandedNodeIdDataSet
```

As we can observe, the topics are separated for the Data and MetaData messages. These topics differ only in one level, which is the MessageType level, with "data" and "metadata". The prefix remains the default "opcua", and the "json" indicates that JSON encoding is used. PublisherId is set as default, followed by the WriterGroup and DataSetWriter.

Configuring the default topic structure with Forge proved to be straightforward, with variables easily added to DataSets and no significant issues encountered during publisher configuration. The challenge lay in determining the strategy, naming conventions, and content for DataSets and WriterGroups to ensure consistency across the system. Currently, there are no established best practices for OPC UA PubSub configurations. Fortunately, this simple setup did not pose significant challenges as the system did not require expansion. Additionally, the default topic structure provided a topic level to separate the Data and MetaData, which was more problematic in the UNS topic structure.

The usage of the default OPC UA topic standardizes a few levels but still allows for user-specific design options that can be utilized to represent certain models.

At the beginning of the topic, there is the option to provide a system-defined name, as recommended in the specification [11]. The so-called Prefix part is typically kept in one level, as splitting the prefix into multiple levels can lead to mismatched specification levels. By default, this part of the topic is set to 'opcua'. The Prefix part of the topic could represent different factories, allowing the end of the topics to remain

the same between factories as they are separated at the first level. This setup used the default value, as additional configuration was not required.

The next few levels of the topic describe the payload, format, and sender. However, the topic also includes two levels inherited from `WriterGroup` and `DataSetWriter`, which can be user-defined without any specific specification. The only recommendation is to use human-readable names. In this setup, the `WriterGroup` describes the object from which the data is sent. The `DataSetWriter` then describes the strategy used for the names of the `DataSet` fields.

6.1.2 UNS topic structure

```
Objects /
  MyHome /
    Bedroom /
      RoomMeasurement
    Livingroom /
      RoomMeasurement
```

This topic structure shows the hierarchical structure of the Address Space. It uses the `BrowseName` of the Nodes in each level.

Configuring the UNS topic structure required additional effort, as it involved setting up more `DataSets` and `DataSetWriters`. However, the planning and naming phase was much simpler as everything had already been established during the creation of the Address Space hierarchy.

As noted in the previous chapter, configuring the `MetaData` message posed a challenge due to the hierarchical topic structure's lack of a distinct topic for it. Consequently, the `MetaData` messages were configured to share the same topics as the data, resulting in them not being visibly separated in the topic tree.

The clear benefit is that the structure of the system can be easily seen from the topic tree. It facilitates easy navigation and filtering to receive messages from different parts of the system.

Another finding was that with the hierarchical structure, the message sizes could not be easily controlled, as the variables are included according to the hierarchy. This could potentially result in larger messages in cases where many variables are grouped under the same object.

Additionally, it was found that the hierarchical structure increased the number of messages being sent, as the variables were not grouped. A notable finding is that users cannot control the volume of messages, as the grouping cannot be adjusted and variables are sent in their messages.

The topic structure plays a critical role in MQTT communication as it allows messages to be filtered based on their content, unlike the `DataSet` field's names. While the hierarchical topic tree provides a clear view of the hierarchy, it lacks support for encoding, message type, and other features. For instance, logically placing `MetaData` within the hierarchical structure can be challenging, highlighting the need for new standards to guide users in constructing topics effectively.

6.2 Message structure

Complete message examples can be found in Appendix C. Overall, the messages look good, although Forge falls short of completely filling the PubSub specification examples.

6.2.1 MetaData messages

MetaData messages are intended to provide information such as engineering units, descriptions, and details about the VariableType. The benefit of this is that subscribers can receive this information from MetaData messages, potentially reducing the size of data messages as they no longer need to include this metadata. However, in Forge's current implementation, there is no option to add custom properties to the MetaData message. Nevertheless, if the Description attribute of the Node is provided, it is included in the message, as shown in the following example:

```
"Description" : {  
    "Text" : "List of all rooms in smart home"  
}
```

The engineering unit would ideally be included in the MetaData message as it does not need to be sent every time alongside the data. It can be assumed that the engineering unit remains constant and does not change continuously.

Currently, the PubSub solution in Forge includes information on the data and value type. There are for example the information of the DataType impressed with values BuiltInType and DataType that can be constructed of Id and Namespace like in the following example:

```
"DataType" : {  
    "Id" : 3003,  
    "Namespace" : 31  
}
```

Already at this point, an issue can be identified. The Namespace value indicates the namespace as an index in the namespace table. For example, the namespace index 31 can be verified from Forge's namespace table as the index for the smart home model. However, it is evident that the subscriber has no access to this namespace table, which is dynamically created for every OPC UA server at startup. This dynamic creation may cause the namespace index to change unpredictably.

BuiltInType is working since it corresponds to basic types that every OPC UA should possess, allowing subscribers to interpret them correctly.

```
"BuiltInType" : 12
```

This example can be verified to indicate the string DataType. It doesn't necessitate namespace information because BuiltInTypes are in the basic namespace, consistently located at index 0 in the namespace table.

Array dimensions are presented in OPC UA with the value ValueRank as follows:

```
"ValueRank" : 1
```

Between the different VariableDataSets, it can be observed that the MetaData messages remain unaffected. Only the identifier changes according to the configuration.

6.2.2 VariableDataSet messages

In the VariableDataSets, the name of the DataSet field could be easily modified using different strategies. The variances and advantages of these modifications are introduced next.

" Lights "

The BrowseName strategy uses the value from the Node's BrowseName attribute. However, this value is not unique in the OPC UA Address Space. It is highly likely to encounter duplicate names for the DataSet fields within a single message, resulting in an inability to distinguish between measurements. While this approach offers a compact and straightforward DataSet field presentation, it may lack sufficient variation when used with a wide DataSetWriter. Additionally, this arrangement fails to provide any contextual information regarding the origin of the measurement. For example, if RoomMeasurement were added from both rooms in this setup, the message would contain identical names for the DataSet fields. Furthermore, only one RoomMeasurement would require additional information for the subscriber to identify the source room. Despite these limitations, this strategy proved useful for the hierarchical topic structure.

"MyHome/ Livingroom / RoomMeasurements / Lights "

The BrowsePathDataSet strategy also utilizes BrowseNames, but in this DataSet, the entire BrowsePath represents the name of a value in the message. This approach provides more information about the value and its context. Additionally, it enhances the uniqueness of the DataSet fields in the Address Space, although the lack of namespace indexes does not guarantee this uniqueness. However, this approach has a drawback as the BrowsePath may become very long, resulting in longer names and larger messages.

It is important to note that while we obtain the path to the variable, we do not have information about the NodeClasses in the path. This limitation is evident in structured variables like RoomMeasurement, where the variable RoomMeasurement creates the same path as the objects in the hierarchy.

" nsu=http://www.prosysopc.com/OPCUA/ Forge ; s=MyHome/
Livingroom / RoomMeasurements / Lights "

The ExpandedNodeIdDataSet is the most informative option among the previous two. In ExpandedNodeId, the namespaceURI is included in string format rather than as an index. Consequently, the subscriber does not need to have the same namespace table to target the correct namespace. Additionally, ExpandedNodeId provides the full NodeId of the Node, allowing the subscriber to identify the specific Node accurately.

However, it's worth noting that the NodeId does not have to be a string or human-readable. Therefore, this option may not be self-explanatory, and a subscriber would require knowledge of the NodeIds to recognize the meaning of the value.

Forge uses the string value of the Enumerator variable. This ensures that the value in the MQTT message is human-readable, and the subscriber does not need to be aware of the specific values for the enumerator to understand the value. This approach is particularly beneficial when the enumerator values are self-explanatory like in the following example from a data message:

```
nsu=http://www.prosysopc.com/OPCUA/Forge;s=MyHome/Mode"
: {
    "Value" : "Home_0"
}
```

Forge generated most of the headers, but it was lacking the DataSetWriterName header. However, the DataSetWriter could still be identified from the topic or by using the DataSetWriterId header. Nonetheless, the Id alone would not provide much information for the subscriber, as it would not have access to match the Id with a corresponding name.

OPC UA Browser has the knowledge to parse the PubSub messages. The following Figure 24 shows how well it could parse the messages:

Data		Events				
PublisherId	DataSetWriterId	Name	Value	StatusCode		
urn:DESKTOP-6ELQEFB:OPCUA:Forge	3	nsu=http://www.prosysopc.com/OPCUA/Forge;s=MyHome/Livingroom/RoomMeasurements/Temperature	22.6	GOOD (0x00000000) "The operation succeeded."		
urn:DESKTOP-6ELQEFB:OPCUA:Forge	3	nsu=http://www.prosysopc.com/OPCUA/Forge;s=MyHome/Livingroom/RoomMeasurements/Lights	false	GOOD (0x00000000) "The operation succeeded."		
urn:DESKTOP-6ELQEFB:OPCUA:Forge	3	nsu=http://www.prosysopc.com/OPCUA/Forge;s=MyHome/Livingroom/RoomMeasurements/Ventilation	89.0	GOOD (0x00000000) "The operation succeeded."		
urn:DESKTOP-6ELQEFB:OPCUA:Forge	3	nsu=http://www.prosysopc.com/OPCUA/Forge;s=MyHome/Rooms	["Bedroom", "Livingroom"]	GOOD (0x00000000) "The operation succeeded."		

Figure 24: PubSub variable messages received by OPC UA Browser.

The structured variable from the smart-home model could not be parsed since the information for that is not accessible for Browser. In the raw log the structured variable was conducted in following object format:

```
"nsu=http://www.prosysopc.com/OPCUA/Forge;s=MyHome/
Bedroom/RoomMeasurements" : {
    "Value" : {
        "Body" : {
            "Lights" : true ,
            "Ventilation" : 30,
            "Temperature" : 24.62373
        }
    }
}
```

6.2.3 EventDataSet messages

EventDataSets are acyclic messages that can be published via MQTT. Essentially, the event properties are parsed into an MQTT message and transmitted when an event occurs. However, the event message faces similar challenges with namespace and NodeIds as the VariableDataSets.

Forge allowed easy configuration for new events. The event properties were easy to configure and include different values. A problem with such a solution is that it is not standardized at all and thus, it provides only a custom solution. Still, it gives a great idea of how the MQTT could be configured if the templates would be for example standardized for specific use cases. The configured event generated the following values to the event message.

```
"Message" : "LightsOn-Event: Light state: \" Lights are on \"",
"Severity" : "100"
```

This was parsed by the OPC UA Browser as seen from Figure 25:

Data		Events				
PublisherId	DataSetWriterId	SourceName	SourceNode	Severity	Message	
urn:DESKTOP-6ELQEFB:OPCUA:Forge	9	Server	{Id: "MyHome/Livingroom", Namespace: "http://www.prosysopc.com/OPCUA/Forge", IdType: 1}	100	LightsOn-Event: Light state: 'Lights are on'	
urn:DESKTOP-6ELQEFB:OPCUA:Forge	9	Server	{Id: "MyHome/Livingroom", Namespace: "http://www.prosysopc.com/OPCUA/Forge", IdType: 1}	100	LightsOn-Event: Light state: 'Lights are on'	
urn:DESKTOP-6ELQEFB:OPCUA:Forge	9	Server	{Id: "MyHome/Livingroom", Namespace: "http://www.prosysopc.com/OPCUA/Forge", IdType: 1}	100	LightsOn-Event: Light state: 'Lights are on'	
urn:DESKTOP-6ELQEFB:OPCUA:Forge	9	Server	{Id: "MyHome/Livingroom", Namespace: "http://www.prosysopc.com/OPCUA/Forge", IdType: 1}	100	LightsOn-Event: Light state: 'Lights are on'	
urn:DESKTOP-6ELQEFB:OPCUA:Forge	9	Server	{Id: "MyHome/Livingroom", Namespace: "http://www.prosysopc.com/OPCUA/Forge", IdType: 1}	100	LightsOn-Event: Light state: 'Lights are on'	
urn:DESKTOP-6ELQEFB:OPCUA:Forge	9	Server	{Id: "MyHome/Livingroom", Namespace: "http://www.prosysopc.com/OPCUA/Forge", IdType: 1}	100	LightsOn-Event: Light state: 'Lights are on'	

Figure 25: PubSub event messages received by OPC UA Browser.

One key benefit of the EventDataSet compared to the VariableDataSet is that it is only sent when an event occurs, as specified. This can be particularly important in environments with a high volume of messages, where processing speed may be affected by every message, such as with data storage.

7 Discussion

In this chapter, a discussion is provided regarding the findings. Emphasis will be placed on the problem of the object model in MQTT messaging, and the benefits of different MQTT configurations will also be considered.

7.1 Topic role in MQTT messaging

Comparing the UNS topic to the OPC UA default topic in this setup revealed that the UNS topic structure generated more messages. This outcome was expected, as the variables needed to be configured to separate DataSets. The optimization between message size and message count can be influenced by processing power and the pricing of communication services. For example, Azure IoT Hub pricing is determined according to message count per day [31]. Therefore, it would be highly advantageous if users could influence the message architecture.

Considering a topic structure that strictly follows the hierarchy of the Address Space, it's not possible to group variables from different locations into one VariableDataSet. While subscribers can subscribe to receive messages from multiple topics using wildcards, the variables remain in separate messages. On the other hand, the default topic structure permits the creation of DataSets that include data from various parts of the system, enabling the design of more advanced DataSets.

The current PubSub specification standardizes the default topic structure, limiting the levels that can be customized to just a few. WriterGroup and DataSetWriter levels are user-defined and can be utilized to establish context for the message structures. With only two levels of hierarchy, it's not possible to cover much of the system's structure. However, these levels could be designed with other methodologies such as maintenance, production reports (e.g. Overall Equipment Effectiveness (OEE) report), functions, and machine measurements. It could be also possible to present the hierarchy in one topic level and use that for the WriterGroup. In this setup, the WriterGroup could be named "MyHome:Livingroom:RoomMeasurement".

On the other hand, the hierarchy-based strategy for the topic tree allows subscribers to create a simple folder structure of the system from which they are receiving messages. This type of structure is easy for humans to navigate and understand.

The option for using the BrowsePath for the DataSet fields' names in messages provided an idea of partially dividing the topic hierarchy into the message itself. This approach would enable grouping an entire object into one message by incorporating the hierarchy to an object in the topic and then the hierarchy to variables inside the object using BrowsePath from the object to the variable in the DataSet field's name.

The default topic had the benefit of providing standardized levels for representing specific information in the messages. For instance, MetaData messages and Data messages could be easily identified from the topic. This feature should be incorporated into the UNS solution to standardize the location for different MQTT messages. For instance, the UNS should be used so that the standard levels are at the beginning like this:

opcua/json/metadata/urn:DESKTOP-6ELQEFB:OPCUA:Forge/Objects/MyHome/Bedroom/RoomMeasurement

In the topic, there exists a level for PublisherId, which is unique to identify the publisher. This limits the possibility of different publishers to publish on the same topic. The PubSub specification is vague in this part, but if this topic structure is used, there is no possibility of having a common topic so multiple publishers could publish data to.

7.2 Message contents

Essentially, information model exchange focuses on a scenario where MQTT communication occurs between two OPC UA systems. It is assumed that both the publisher and subscriber would benefit from the OPC UA information models.

In OPC UA, NodeId plays a crucial role in identifying the structure of Nodes. As outlined in this thesis, a NodeId consists of a namespace index and an ID. This mechanism works well for basic client-server communication, where the client has access to the namespace table and the entire Address Space. However, in PubSub, NodeId serves not only to identify instances but also as an essential attribute for identifying types. Considering type identification, it prompted the idea of using NodeSet2.xml files to provide subscribers with information about the types. While this approach is feasible, there are uncertainties regarding the handling of various versions of information models.

Even though it could be considered that the cloud library could assist the subscriber in understanding the type by fetching type information via a REST API, the subscriber would still lack knowledge of the instances. Of course, it is not mandatory for the subscriber to know the entire Address Space to benefit from type information. Just emphasizing that it's worth noting that while type information obtained from the cloud library could be used to identify the type of the received Variable, it does not encompass the entire model to which the Variable belongs.

From the setup that was performed, it was observed that ExpandedNodeId could be utilized for the DataSet fields in MQTT messages, ensuring that duplicate DataSet fields would not be included in the MQTT message. However, it was also noted that ExpandedNodeId might be lengthy and does not necessarily provide a human-readable name. On the contrary, ExpandedNodeId ensures an unambiguous method to search for the Node from the source server. For instance, a maintenance system receiving data via MQTT might need to relay information back to the source Node.

A comparison between the object version and the variable version of the person type revealed that configuring the variable was much simpler than configuring the PubSub message. It was neatly grouped into the message. However, it was noted that the Browser could not parse the custom VariableType from the message.

In addition, it is evident that exchanging the entire model through MQTT would be very challenging. Information about ObjectTypes, ReferenceTypes, and similar elements is completely absent from the messaging.

7.3 Forge's capabilities

Forge enables PubSub communication with an easy and modern configuration interface. However, there are some missing parts in the messages generated by Forge. During the setup, it was observed that Forge's PubSub did not include a namespace table in the MetaData messages. Analyzing the concept of a namespace table within MQTT messages reveals its potential complexity. For instance, a subscriber may receive multiple MetaData messages, each with its own namespace table, leading to potential discrepancies. Additionally, information such as BrowsePath include a namespace index referencing the source server namespace table, which would require conversion to the MQTT message namespace table or access to the source namespace table by the subscriber.

Additionally, it was found that Forge did not include parsing instructions for the custom variable in the MetaData message. These instructions are necessary for an OPC UA subscriber to parse the object from the MQTT message if it does not know the type.

Also, the MetaData message could not be modified by any means, meaning that adding the properties from objects or variables could not be done. Only the description was included by default in the MetaData messages. In addition to properties, it was considered that MetaData message could contain other attributes of a Node as well. For example, for maintenance usage, it is important that the source for the data could be identified unambiguously when maintenance actions are required.

In the end, Forge supported both VariableDataSets and EventDataSets, and the messages were generated successfully. The message structures mostly followed the OPC UA specification. It could be verified that the OPC UA Browser could parse variables from the MQTT messages.

7.4 Future directions

MQTT communication is a hot topic at the moment and OPC Foundation is also focusing on solving the details to include OPC UA information models in MQTT messaging. One example of this is the new PubSub specification which was released in December 2023. OPC Foundation should stay active and try to provide proper examples to help others utilize PubSub. The PubSub specification is 225 pages and extremely heavy for users to adopt and thus for example Sparkplug might be a more compelling architecture with its simplicity for use cases where client-server communication is not necessary.

The concept of the cloud library also requires a comprehensive example. While the idea holds promise and could help in exchanging type information in PubSub communication, it does not address the challenge of receiving knowledge about instances in the Address Space. Adding instances into the cloud library could potentially address this issue.

OPC UA PubSub could expand upon the default topic tree by instructing the usage of other topic structures like the UNS topic structure. This would be particularly advantageous for use cases where the Address Space hierarchy plays a crucial role.

The UNS topic structure would gain from standardized topic levels similar to those found in the current default topic tree in PubSub.

Standardized MQTT message contents derived from specific information models could help in the implementation of PubSub communication. Even though this addition would expand the OPC UA specification further, with appropriate structuring, it could serve as an excellent starting point for utilizing PubSub. The DataSetClass concept covers this aspect but leaves it somewhat open to how to do the implementation. Moreover, including message topics within these standardized MQTT messages could help to understand the usage of the topic levels.

Additionally, in this experiment, it was observed that the Forge application offers a straightforward configuration interface for PubSub communication. A notable improvement that Forge lacked was the ability to easily include the properties of Variables in MQTT messages. The PubSub configurations should have options for, which attributes and properties of Variables are included in the PubSub messages.

8 Conclusion

This thesis has presented a comprehensive exploration of evolving OPC UA specification and the integration of OPC UA Information Models with the PubSub over MQTT communication protocol within the field of industrial automation. The study introduced OPC UA's extensive information modeling capabilities and assessed their practical application in MQTT environments through the use of Prosys OPC UA Forge.

It was discussed that the PubSub communication model is valuable for OPC UA and the integration of OPC UA Information Models with MQTT enhances the interoperability and efficiency of industrial automation systems. The new version 1.05.03 of the PubSub specification offers a great deal of instructions for the topic structure and a few example messages.

Prosys OPC UA Forge proves to be a valuable tool in configuring OPC UA publishers, facilitating seamless PubSub communication over MQTT. The experimental setup demonstrated the potential of OPC UA's robust information modeling in real-world MQTT applications, bridging the gap between OT and IT systems.

The complexity of the OPC UA standard presents a steep learning curve and thus adaption requires clearer implementation guidelines and examples. It is important to maintain focus on user needs and make the features easier to adapt. Further development is required to fully realize the integration of extensive information modeling into MQTT messaging.

In conclusion, the research conducted in this thesis contributes to the ongoing efforts to achieve a digitally transformed industrial landscape. The successful integration of OPC UA Information Models with MQTT communication models holds promise for the future of industrial automation, offering a pathway toward a more connected, efficient, and intelligent manufacturing environment.

References

- [1] OPC Foundation, “Opc 10000-1 – part 1: Overview and concepts”, en, version 1.05.02, 2022.
- [2] OPC Foundation, “Opc 10000-3 – part 3: Address space model”, en, version 1.05.03, 2023.
- [3] M. Graube, S. Hensel, C. Iatrou, and L. Urbas, “Information models in opc ua and their advantages and disadvantages”, in *2017 22nd IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*, IEEE, 2017, pp. 1–8.
- [4] P. Drahoš, E. Kučera, O. Haffner, and I. Klimo, “Trends in industrial communication and opc ua”, in *2018 Cybernetics & Informatics (K&I)*, 2018, pp. 1–5. DOI: [10.1109/CYBERI.2018.8337560](https://doi.org/10.1109/CYBERI.2018.8337560).
- [5] T. Borangiu, D. Trentesaux, A. Thomas, P. Leitão, and J. Barata, *Digital transformation of manufacturing through cloud services and resource virtualization*, 2019.
- [6] P. Helo, M. Suorsa, Y. Hao, and P. Anussornnitisarn, “Toward a cloud-based manufacturing execution system for distributed manufacturing”, *Computers in industry*, vol. 65, no. 4, pp. 646–656, 2014.
- [7] L. Beňo, R. Pribiš, and R. Leskovský, “Processing data from opc ua server by using edge and cloud computing”, *IFAC-PapersOnLine*, vol. 52, no. 27, pp. 240–245, 2019, 16th IFAC Conference on Programmable Devices and Embedded Systems PDES 2019, ISSN: 2405-8963. DOI: <https://doi.org/10.1016/j.ifacol.2019.12.645>. [Online]. Available: <https://www.sciencedirect.com/science/article/pii/S240589631932590X>.
- [8] OPC Foundation, “Opc 10000-2: Ua part 2: Security”, en, version 1.05.03, 2023.
- [9] OPC Foundation, “Opc 10000-4 – part 4: Services”, en, version 1.05.03, 2023.
- [10] OPC Foundation, “Opc 10000-5 – part 5: Information model”, en, version 1.05.03, 2023.
- [11] OPC Foundation, “Opc 10000-14 – pubsub”, en, version 1.05.03, 2023.
- [12] “OPC UA Online Reference - Released Specifications”. (2024), [Online]. Available: <https://reference.opcfoundation.org/> (visited on 07/04/2024).
- [13] “OPC Foundation Home page”. (2024), [Online]. Available: <https://opcfoundation.org/> (visited on 04/07/2024).
- [14] OPC Foundation, “Opc 10000-8 – part 8: Dataaccess”, en, version 1.05.03, 2023.
- [15] OPC Foundation, “Opc 10000-11 – part 11: Historical access”, en, version 1.05.03, 2023.

- [16] OPC Foundation, “Opc 10000-9 – part 9: Alarms and conditions”, en, version 1.05.03, 2023.
- [17] W. Mahnke, S. Leitner, and M. Damm, *OPC Unified Architecture* (SpringerLink: Springer). Springer Berlin Heidelberg, 2009, 399 pp., ISBN: 9783540688990.
- [18] “UA CloudLibrary”. (2024), [Online]. Available: <https://opcfoundation.org/markets-collaboration/cloudlib/> (visited on 04/07/2024).
- [19] Standard, OASIS, “MQTT Version 3.1.1”, *Retrieved June*, 2014.
- [20] HiveMQ, *MQTT & MQTT 5 Essentials*. HiveMQ, 2020, 72 pp., ISBN: 978-3-00-067913-1. [Online]. Available: <https://www.hivemq.com/download-mqtt-ebook/> (visited on 04/07/2024).
- [21] HiveMQ, *MQTT Sparkplug Essentials*. HiveMQ, 2021, 18 pp. [Online]. Available: <https://www.hivemq.com/download-sparkplug-ebook/> (visited on 04/07/2024).
- [22] HiveMQ, *Unified Namespace (UNS) Essentials*. HiveMQ, 24 pp. [Online]. Available: <https://www.hivemq.com/mqtt/unified-namespace-uns-essentials-iiot-industry-40/> (visited on 04/07/2024).
- [23] Standard, OASIS, “MQTT Version 5.0”, *Retrieved June*, 2019.
- [24] “UaModeler”. (2024), [Online]. Available: <https://www.unified-automation.com/products/development-tools/uamodeler.html> (visited on 04/07/2024).
- [25] “Unified Automation”. (2024), [Online]. Available: <https://www.unified-automation.com/index.html> (visited on 04/07/2024).
- [26] “Prosyst OPC UA Simulation server”. (2024), [Online]. Available: <https://prosysopc.com/products/opc-ua-simulation-server/> (visited on 04/07/2024).
- [27] “Prosyst OPC Ltd”. (2024), [Online]. Available: <https://prosysopc.com/> (visited on 04/07/2024).
- [28] “Prosyst OPC UA Forge”. (2024), [Online]. Available: <https://prosysopc.com/products/opc-ua-forge/> (visited on 04/07/2024).
- [29] “Eclipse Mosquitto”. (2024), [Online]. Available: <https://mosquitto.org/> (visited on 04/07/2024).
- [30] “Prosyst OPC UA Browser”. (2024), [Online]. Available: <https://prosysopc.com/products/opc-ua-browser/> (visited on 04/07/2024).
- [31] “Azure IoT Hub pricing”. (2024), [Online]. Available: <https://azure.microsoft.com/en-us/pricing/details/iot-hub/> (visited on 04/04/2024).

Appendix A

JSON Data message description

Please note that the data types enclosed in brackets in the keys are not part of the actual messages.

Listing 1: Data Messages explained.

```
DataSetMessage = { #NetworkMessage
  "MessageId(String)": "Globally unique identifier for
the message (Guid in string format). Always present
if network headers flag is set to true.",

  "MessageType(String)": "ua-data/ua-application/
ua-endpoints/ua-status/ua-connection. Always present
if network headers flag is set to true.",

  "PublisherId(String)": "PublisherId from the
PubSubConnection",

  "WriterGroupName(String)": "Name of the WriterGroup",

  "DataSetClassId(Guid)": "DataSet can be constructed
according to a DataSetClass. This Id is Guid to
identify the class.",

  "Messages": [
    { #DataSetMessage starts here
      "DataSetWriterId(UInt16)": "Unique in the
context of a Publisher.",

      "DataSetWriterName(String)": "Name of the
DataSetWriter.",

      "PublisherId(String)": "PublisherId. Shall
be omitted if PublisherId is present in
NetworkMessage",

      "WriterGroupName(String)": "WriterGroupName.
Shall be omitted if WriterGroupName is present
in NetworkMessage",

      "SequenceNumber(UInt32)": "The SequenceNumber
starts at 0 and shall be incremented by exactly
one for each message.",
```

"MetaDataVersion(ConfigurationDataType)": "Version of the DataSetMetaData describing the DataSetMessage",

"MinorVersion(VersionType)": "MinorVersion of the DataSetMetaData describing the DataSetMessage. Shall be omitted if the MetaDataVersion is contained in the datasetmsgheader",

"Timestamp(DateTime)": "The time when DataSetMessage was created.",

"Status(StatusCode)": "The overall status of the DataSetMessage.",

"MessageType(String)": "ua-keyframe/ua-deltaframe/ua-event/ua-keepalive",

```
"Payload(object)": [
  { #DataSet Fields
    "Name_of_the_value": {
      "value": "<value>",
      "SourceTimestamp": "<SourceTimestamp>",
      "ServerTimestamp": "<ServerTimestamp>",
      "SourcePicoSeconds": "<SourcePicoSeconds>
      (This flag is ignored if the SourceTimestamp
      flag is not set.)",
      "ServerPicoSeconds": "<ServerPicoSeconds>
      (This flag is ignored if the ServerTimestamp
      flag is not set.)",
      "Status": {
        "Code": "",
        "Symbol": ""
      }
    }
  }
]
```

Appendix B

JSON MetaData message description

Listing 2: MetaDataMessage explained.

```
MetaDataMessage = { #DataSetMetaData
  "MessageId(String)": "A globally unique identifier for
the message. This value is mandatory. ",
  "MessageType(String)": "ua-metadata",
  "PublisherId(String)": "PublisherId from the
PubSubConnection",
  "DataSetWriterId(UInt16)": "Unique in in the context
of a Publisher.",
  "DataSetWriterName(String)": "Name of the
DataSetWriter. This is only header in MetaDataMessage
that depends of the JsonDataSetMessageContentMask",
  "TimeStamp(UtcTime)": "When the message was first sent
to the middleware.",
  "MetaData(DataSetMetaDataType)": {
    "Name(String)": "Name of the DataSet",
    "Description(LocalizedText)": "Description of the
DataSet. Default value is null or empty
LocalizedText.",
    "Namespaces(String[])": [
      "Namespaces that are not OPC UA basics but used
in message"
    ],
    "DataSetClassId(Guid)": "If the DataSet is based
on DataSetClass, this shall match the concrete
DataSetClassId.",
    "ConfigurationVersion
(ConfigurationVersionDataType)":
    {
```

```

"MajorVersion": "The configuration version for the
current DataSet configurations",

"MinorVersion": "The configuration version for the
current DataSet configurations"
},

"structureDataTypes(StructureDescription[])/
enumDataTypes(EnumDescription[])/
simpleDataTypes(SimpleTypeDescription[])": [
{
  "DataTypeId": "NodeId of custom Type",

  "Name": "Name of the custom type.",

  "StructureDefinition": {
    "DefaultEncodingId(NodeId)": "The NodeId of
the default DataTypeEncoding for the DataType.
Default shall always be Default Binary
encoding. In nested Structures and not
contained in an Extensin Object use null",

    "BaseDataType(NodeId)": "The NodeId of the
direct supertype of the DataType.",

    "StructureType(StructureType)": "Number
between 0 – 4. 0 – structure ,
1 – Structure with optional fields ,
2 – Union, 3 – Structure with subtyped
values, 4 – Union with subtyped values",

    "Fields(StructureField[])": [
      {
        "Name(String)": "A name for the field that
is unique wihtin the StructureDefinition",

        "Description(LocalizedText)": "Description
of the field",

        "DataType(NodeId)": "The NodeId of the
DataType",

        "ValueRank(Int32)": "The value rank for
the field.",
      }
    ]
  }
}

```

```

    "ArrayDimensions(UInt32[])": [
      "Maximum supported length of each
      dimension. Null if ValueRank <=0. 0
      is maximum."
    ],

    "MaxStringLength(UInt32)": "Maximum
    supported length for text in bytes.
    Maximum length is 0 which is unknown.
    If DataType is not text this is 0.",

    "IsOptional(Boolean)": "True means this
    structure field might be present. Needs to
    follow the StructureType value."
  }
]
}
}
],

"Fields(FieldMetaData)": [
  {
    "Name(String)": "Name of the field which is
    unique in the context of DataSet.",

    "Description(LocalizedText)": "Description",
    "FieldFlags(DataSetFieldFlags)": "field_flags",
    "BuiltInType(Byte)": "The built-in data type of
    the field. Values defined in OPC 10000-6.",

    "DataType(NodeId)": "The NodeId of the DataType.
    For special cases the semantics in structure/
    enum-DataType field.",

    "ValueRank(Int32)": "n>0 array with n
    dimensions, 0 array with one or more dimensions,
    -1 scalar, -2 scalar or array, -3 scalar or one
    dimension.",

    "ArrayDimensions(UInt32[])": [
      "Maximum supported length of each dimension.
      Null if ValueRank <=0. 0 is maximum."
    ],

    "MaxStringLength(UInt32)": "Maximum supported

```

length for text in bytes. Maximum length is 0 which is unknown. If DataType is not text this is 0.",

"DataSetFieldId(Guid)": "The unique ID for the field in the DataSet. The ID is generated when the field is added to the list. A change of the position of the field in the list shall not change the ID. ",

```
"Properties(KeyValuePair[])": [  
    "Additional semantics for example engineering  
    unit."  
]  
}  
]  
}  
}
```

Listing 3: NodeId's and LocalizedText's structures.

```
NodeId = {  
    "IdType(Enum)": "0 numeric , 1 string ,  
    2 Guid , 3 opaque.",  
  
    "Id": "Identifier in format of IdType.",  
  
    "Namespace(UInt16)": "number in the  
    namespace table",  
}  
  
LocalizedText = {  
    "Locale(LocaleId)": "The identifier for the locale  
    (e.g. en-US).",  
  
    "Text(String)": "description"  
}
```

Appendix C

Full MQTT Data and MetaData messages generated by Forge

Listing 4: Forge generated Data message

```
03/30/24 21:55:04.0310000 EET
opcua/json/data/urn:DESKTOP-6ELQEFB:OPCUA:Forge/MyHome/
MyHomeExpandedNodeIdDataSet
{
  "MessageId": "65cbfe92-5a2c-432b-bf1c-90b7fd11d0df",
  "MessageType": "ua-data",
  "PublisherId": "urn:DESKTOP-6ELQEFB:OPCUA:Forge",
  "Messages": [ {
    "DataSetWriterId": 3,
    "MessageType": "ua-keyframe",
    "Payload": {
      "nsu=http://www.prosysopc.com/OPCUA/Forge;s=MyHome
/Bedroom/RoomMeasurements": {
        "Value": {
          "Body": {
            "Lights": true,
            "Ventilation": 30,
            "Temperature": 24.62373
          }
        }
      }
    },
    "nsu=http://www.prosysopc.com/OPCUA/Forge;s=MyHome
/Rooms": {
      "Value": [ "Bedroom", "Livingroom" ]
    },
    "nsu=http://www.prosysopc.com/OPCUA/Forge;s=MyHome
/Livingroom/RoomMeasurements/Lights": {
      "Value": true
    },
    "nsu=http://www.prosysopc.com/OPCUA/Forge;s=MyHome
/Livingroom/RoomMeasurements/Temperature": {
      "Value": 21.9
    },
    "nsu=http://www.prosysopc.com/OPCUA/Forge;s=MyHome
/Livingroom/RoomMeasurements/Ventilation": {
      "Value": 30
    },
    "nsu=http://www.prosysopc.com/OPCUA/Forge;s=MyHome
/Mode": {
```

```

        "Value": "Home_0"
    }
}
} ]
}

```

Listing 5: Forge generated MetaData message for data message

```

03/30/24 21:56:04.6770000 EET
opcua/json/metadata/urn:DESKTOP-6ELQEFB:OPCUA:Forge/
MyHome/MyHomeExpandedNodeIdDataSet
{
  "MessageId": "c854c73b-6a4a-4388-9461-d3728b309757",
  "MessageType": "ua-metadata",
  "PublisherId": "urn:DESKTOP-6ELQEFB:OPCUA:Forge",
  "DataSetWriterId": 3,
  "MetaData": {
    "Name": "ExpandedNodeIdDataSet",
    "Fields": [ {
      "Name": "nsu=http://www.prosysopc.com/OPCUA/
Forge;s=MyHome/Bedroom/RoomMeasurements",
      "Description": {
        "Text": "Gather measurements from the room."
      },
      "BuiltInType": 22,
      "DataType": {
        "Id": 3003,
        "Namespace": 31
      },
      "ValueRank": -1,
      "DataSetFieldId": "328656a2-e40d-4578-b0dc-c92bbece5321"
    }, {
      "Name": "nsu=http://www.prosysopc.com/OPCUA/Forge;s=MyHome/
Rooms",
      "Description": {
        "Text": "List of all rooms in smart home"
      },
      "BuiltInType": 12,
      "DataType": {
        "Id": 12
      },
      "ValueRank": 1,
      "DataSetFieldId": "15ce79dc-d53d-4ab1-82a5-066fedae73d5"
    }, {
      "Name": "nsu=http://www.prosysopc.com/OPCUA/Forge;s=MyHome/
Livingroom/RoomMeasurements/Lights",

```

```

    "Description": { },
    "BuiltInType": 1,
    "DataType": {
      "Id": 1
    },
    "ValueRank": -1,
    "DataSetFieldId": "56332e4c-3b38-4da4-9875-e197f1b6fb47 "
  }, {
    "Name": "nsu=http://www.prosysopc.com/OPCUA/Forge;s=MyHome/Livingroom/RoomMeasurements/Temperature",
    "Description": { },
    "BuiltInType": 11,
    "DataType": {
      "Id": 11
    },
    "ValueRank": -1,
    "DataSetFieldId": "e42f4ab3-7156-4ddb-bebf-ed26c7a9904a "
  }, {
    "Name": "nsu=http://www.prosysopc.com/OPCUA/Forge;s=MyHome/Livingroom/RoomMeasurements/Ventilation",
    "Description": {
      "Text": "Ventilation speed in percentage."
    },
    "BuiltInType": 4,
    "DataType": {
      "Id": 4
    },
    "ValueRank": -1,
    "DataSetFieldId": "a3d632b7-a1f9-43f4-8d22-ae73afc4aec2 "
  }, {
    "Name": "nsu=http://www.prosysopc.com/OPCUA/Forge;s=MyHome/Mode",
    "Description": {
      "Text": "In which mode the smart home is currently."
    },
    "BuiltInType": 24,
    "DataType": {
      "Id": 3004,
      "Namespace": 31
    },
    "ValueRank": -1,
    "DataSetFieldId": "327081e6-828c-4d2a-9b49-66a4b9d1c9b7 "
  } ]
}
}

```

Listing 6: Forge generated Event data message

```
03/30/24 21:57:40.0310000 EET
opcua/json/data/urn:DESKTOP-6ELQEFB:OPCUA:Forge/MyHome/
MyHomeLivingroomLightsOn
{
  "MessageId": "b074cb1f-fc9d-4e16-ad95-d43de0199303",
  "MessageType": "ua-data",
  "PublisherId": "urn:DESKTOP-6ELQEFB:OPCUA:Forge",
  "Messages": [ {
    "DataSetWriterId": 9,
    "MessageType": "ua-event",
    "Payload": {
      "SourceName": "Server",
      "SourceNode": {
        "IdType": 1,
        "Id": "MyHome/Livingroom",
        "Namespace": "http://www.prosysopc.com/OPCUA/Forge"
      },
      "Message": "LightsOn-Event: Light state: \"Lights
are on\"",
      "Severity": "100"
    }
  } ]
}
```

Listing 7: Forge generated MetaData message for event data message

```
03/30/24 21:56:34.6760000 EET
opcua/json/metadata/urn:DESKTOP-6ELQEFB:OPCUA:Forge/MyHome/
MyHomeLivingroomLightsOn
{
  "MessageId": "2ff2a6a3-4eea-4c0e-a59c-fb85c0e1064b",
  "MessageType": "ua-metadata",
  "PublisherId": "urn:DESKTOP-6ELQEFB:OPCUA:Forge",
  "DataSetWriterId": 9,
  "MetaData": {
    "Name": "MyHomeLivingroomLightsOn",
    "Fields": [ {
      "Name": "SourceName",
      "Description": { },
      "BuiltInType": 12,
      "DataType": {
        "Id": 12
      },
      "ValueRank": -1,
    } ]
  }
}
```

```

    "DataSetFieldId": "7797443f-7b1a-454e-a90b-80b8ca11694e"
  }, {
    "Name": "SourceNode",
    "Description": { },
    "BuiltInType": 17,
    "DataType": {
      "Id": 17
    },
    "ValueRank": -1,
    "DataSetFieldId": "28beeb38-6205-44d2-881f-972af79c4281"
  }, {
    "Name": "Message",
    "Description": { },
    "BuiltInType": 21,
    "DataType": {
      "Id": 21
    },
    "ValueRank": -1,
    "DataSetFieldId": "86047e6a-9b10-46f5-99db-ef6844b6249c"
  }, {
    "Name": "Severity",
    "Description": { },
    "BuiltInType": 5,
    "DataType": {
      "Id": 5
    },
    "ValueRank": -1,
    "DataSetFieldId": "da6f5a44-f45d-4473-b871-dea32e4414f2"
  } ]
}
}

```